

# Variables and Environments

Principles of Programming Languages

CAS CS 320

Lecture 16

# Practice Problem

$$[(\lambda x. xy)/z](\lambda x. \lambda y. yz)$$

Perform the following substitution, avoiding variable capture.

$$[v/y]x = \begin{cases} v & x = y \\ x & \text{else} \end{cases}$$

$$[v/y](\lambda x. e) = \begin{cases} \lambda x. e & x = y \\ \lambda z. [w/z][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x. [v/y]e & \text{else} \end{cases}$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2)$$

**Answer**

$$[(\lambda x. xy)/z](\lambda x. \lambda y. yz)$$

$$[\lambda x. xy / z](\lambda x. \lambda y. y z) =$$

$$\text{fv}(\lambda x. xy) = \{y\}$$

$$\lambda x. [\lambda x. xy / z](\lambda y. yz) =$$

$$\lambda x. \lambda w. [\lambda x. xy / z][w / y](y z) =$$

$$\lambda x. \lambda w. [\lambda x. xy / z](w z) =$$

$$\lambda x. \lambda w. w(\lambda x. xy)$$

$$\text{fun } x \rightarrow \text{fun } w \rightarrow w(\text{fun } x \rightarrow xy)$$

# Outline

Demo an implementation of the lambda calculus

Discuss the difference between lexical and dynamic scoping

Look at the semantics of variable binding, with examples using environments

Begin to look at how closures are used to implement lexical scoping with environments

# Learning Objectives

- Describe the difference between dynamic and lexical scoping
- Give a sequence of reductions given a toy language which implements dynamic scoping
- Give a semantic derivation in the lambda calculus with let-expressions

# Demo (The Lambda Calculus)

# Variables

---

# Two Major Concerns

1. Are variables *mutable*? Can we change their values? Are there restrictions to when we can change the value of a variable?

2. How are variables *scoped*? Dynamically or lexically? Does a binding define its own scope? Is it defined in a block?

OCaml variables are:

- » immutable
- » binding defined
- » lexically scoped

imperative  $\approx$  mutable  
function  $\approx$  immutable

lexical : pretty much  
everything

dynamic : Bash



# Mutability

↳ shadowing

```
let x = 0
let f () =
  let x = 1 in
  ()
print_int x
```

Immutable (OCaml)

```
x = 0
def f():
    global x
    x = 1
print(x)
```

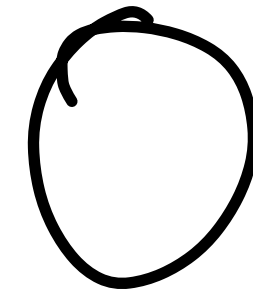
Mutable (Python)

**Definition.** (*informal*) A variable is **mutable** if we are allowed to change its value after it has been declared.

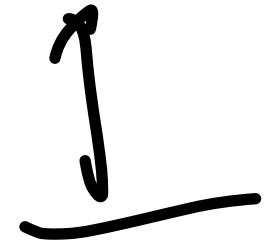
We think of variables as:

- » **names** if they're immutable
- » **(abstract) memory locations** when they're ~~mutable~~ mutable

OCaml:



Python:



# Scope


**Definition.** (*Informal*) The **scope** of a variable binding is *when* and *where* a variable can be accessed

Scoping rules describe how the scope of bindings works in a program.

There are two standard paradigms:

- » dynamic scoping
- » lexical scoping (static scoping)

**Warning.** Scope is one of the most unclear terms in computer science, we might be talking about:

- » the scope of a variable
- » the scope of a binding 
- » the scope of a function
- » scopes in general (like *global* scope)
- » "this variables is not in scope..."

var + its value

# Dynamic Scoping

func def. assign x to 23

```
f() { x=23; g; }  
g() { y=$x; }  
f  
echo $y
```

Bash

print value of y

assign y to value of x

**Dynamic scoping** refers to when bindings are determined at runtime based on *computational context*

This is a *temporal view*, i.e., what a computation done beforehand which affected the value of a variable

# Lexical Scoping

*scope of binding*

```
x = 0
def f():
    x = 1
    return x
assert(f() == 1)
assert(x == 0)
```

Python

```
let x = 0
let f () =
  let x = 1 in
  x
let _ = assert (f () = 1)
let _ = assert (x = 0)
```

OCaml

Lexical (static) scoping refers to the use of textual delimiters to define the scope of a binding

There are two common ways lexical scope is determined:

- » The binding defines its own scope (**let-bindings**)
- » A block defines the scope of a variable (**python functions**)

# Environments

---

# High-Level

$$\{x \mapsto v, y \mapsto w, z \mapsto f\}$$

An *environment* is a data structure which maintains mappings of variables to values

**Terminology.** We call the individual mappings of variables to values **variable bindings**.

Usually it's implemented as an association list or a Map in OCaml. We have a special data structure called **env** for implementing environments.

**The idea.** We will evaluate expressions *relative* to an environment

env:

$$\{x \mapsto 2, y \mapsto 3\}$$

exp:

$$x + y$$

let  $x = 2$  in

let  $y = 3$  in

$$x + y$$

# Operations

Math

OCaml

$\mathcal{E}$

env

$\mathcal{E}[x \mapsto v]$

add x v env

$\mathcal{E}(x)$

find\_opt x env

$\mathcal{E}(x) = \perp$

find\_opt x env = None

Most important operations on environments are the same that are useful for any dictionary-like data structure

**Important:** Adding mappings shadows existing mappings:

$$\mathcal{E}[x \mapsto v][x \mapsto w] = \mathcal{E}[x \mapsto w]$$

$\mathcal{E}$  with  $x$  bound to  $v$

value of  $x$  in  $\mathcal{E}$

$x$  is not bound in  $\mathcal{E}$

$$\{x \mapsto v\} [x \mapsto w] =$$

$$\{x \mapsto w\}$$

# Dynamic Scoping

---



# Toy Language (Syntax)

$\langle \text{prog} \rangle ::= \{ \langle \text{stmt} \rangle; \}$   
 $\langle \text{stmt} \rangle ::= \langle \text{var} \rangle ( ) \{ \langle \text{stmt1} \rangle; \}$  *fun def.*  
 $\langle \text{stmt1} \rangle ::= \langle \text{var} \rangle \mid \langle \text{var} \rangle = \$\langle \text{var} \rangle \mid \langle \text{var} \rangle = \langle \text{num} \rangle$  *assignments*  
 $\langle \text{var} \rangle ::= \dots$   
 $\langle \text{num} \rangle ::= \dots$

This is a small grammar for a language with, numbers, subroutines, and variable assignments

It's like a tiny fragment of Bash

Handwritten code examples illustrating the syntax:

```
f ( ) {  
    x = 23;  
    g;  
};  
g ( ) {  
    y = $x  
};  
f;
```

The examples are annotated with colored boxes: a purple box around the function name 'f', a green box around the opening brace '{', an orange box around the function body, and a red box around the entire function definition 'f ( ) { ... };'. The second example 'g ( ) { ... };' is also annotated with a green box around the opening brace and an orange box around the body.

# Toy Language (Semantics)

```

<prog> ::= { <stmt>; }
<stmt> ::= <var>() { { <stmt1>; } }
          | <stmt1>
<stmt1> ::= <var> | <var> = $<var>
          | <var> = <num>
<var>   ::= ...
<num>   ::= ...
    
```

$$\frac{\text{env} \quad \text{prog}}{\langle \mathcal{E}, f() \{ P \}; Q \rangle \longrightarrow \langle \mathcal{E}[f \mapsto P], Q \rangle} \quad (\text{func}) \quad \text{"put } f \text{ in env"}$$

$$\frac{\text{config} \quad \mathcal{E}(f) = P \in \mathbb{F}}{\langle \mathcal{E}, f; Q \rangle \longrightarrow \langle \mathcal{E}, P Q \rangle} \quad (\text{call}) \quad \text{"prepend body of } f \text{ to env prog"}$$

$$\frac{\mathcal{E}(y) = n \in \mathbb{Z}}{\langle \mathcal{E}, x = \$y; Q \rangle \longrightarrow \langle \mathcal{E}[x \mapsto n], Q \rangle} \quad (\text{aVar})$$

$$\frac{}{\langle \mathcal{E}, x = n; Q \rangle \longrightarrow \langle \mathcal{E}[x \mapsto n], Q \rangle} \quad (\text{aNum})$$

assignments

**Note.** The environment contains both functions ( $\mathbb{F}$ ) and numbers ( $\mathbb{Z}$ )

# Toy Language (Example)

$f() \{ x=23; g; \}; g() \{ y=\$x; \}; f;$

$$\langle \emptyset, f() \{ x=23; g; \} g() \{ y=\$x; \}; f; \rangle \xrightarrow{\quad} \langle E, x = n; Q \rangle \xrightarrow{\quad} \langle E[x \mapsto n], Q \rangle \quad (\text{aNum})$$

$$\langle \{ f \mapsto x=23; g; \}, g() \{ y=\$x; \}; f; \rangle \rightarrow$$

$$\langle \{ f \mapsto \dots, g \mapsto y=\$x \}, f; \rangle \rightarrow$$

$$\langle \{ f \mapsto \dots, g \mapsto \dots \}, x=23; g; \rangle \rightarrow$$

$$\langle \{ x \mapsto 23, f \mapsto \dots, g \mapsto \dots \}, g; \rangle \rightarrow$$

$$\langle \{ x \mapsto 23, f \mapsto \dots, g \mapsto \dots \}, y=\$x; \rangle \rightarrow$$

$$\langle \{ x \mapsto 23, y \mapsto 23, f \mapsto \dots, g \mapsto \dots \}, \epsilon \rangle \quad \checkmark$$

$$\frac{}{\langle E, f() \{ P \}; Q \rangle \rightarrow \langle E[f \mapsto P], Q \rangle} \quad (\text{func})$$

$$\frac{\mathcal{E}(f) = P \in \mathbb{F}}{\langle E, f; Q \rangle \rightarrow \langle E, P Q \rangle} \quad (\text{call})$$

$$\frac{\mathcal{E}(y) = n \in \mathbb{Z}}{\langle E, x = \$y; Q \rangle \rightarrow \langle E[x \mapsto n], Q \rangle} \quad (\text{aVar})$$

$$\frac{}{\langle E, x = n; Q \rangle \rightarrow \langle E[x \mapsto n], Q \rangle} \quad (\text{aNum})$$

# The Takeaway

Defining and implementing dynamic scoping is very easy

It's like maintaining *only* a global scope

But it's behavior is harder to track, and arguably more error prone, you have to *remember* if there is some computation which will put a variable into scope.

**Most modern programming languages implement lexical scoping**

# Lexical Scoping

---

# Didn't we do this?

let x = v in ...

big expr

only 1 or 2 x's

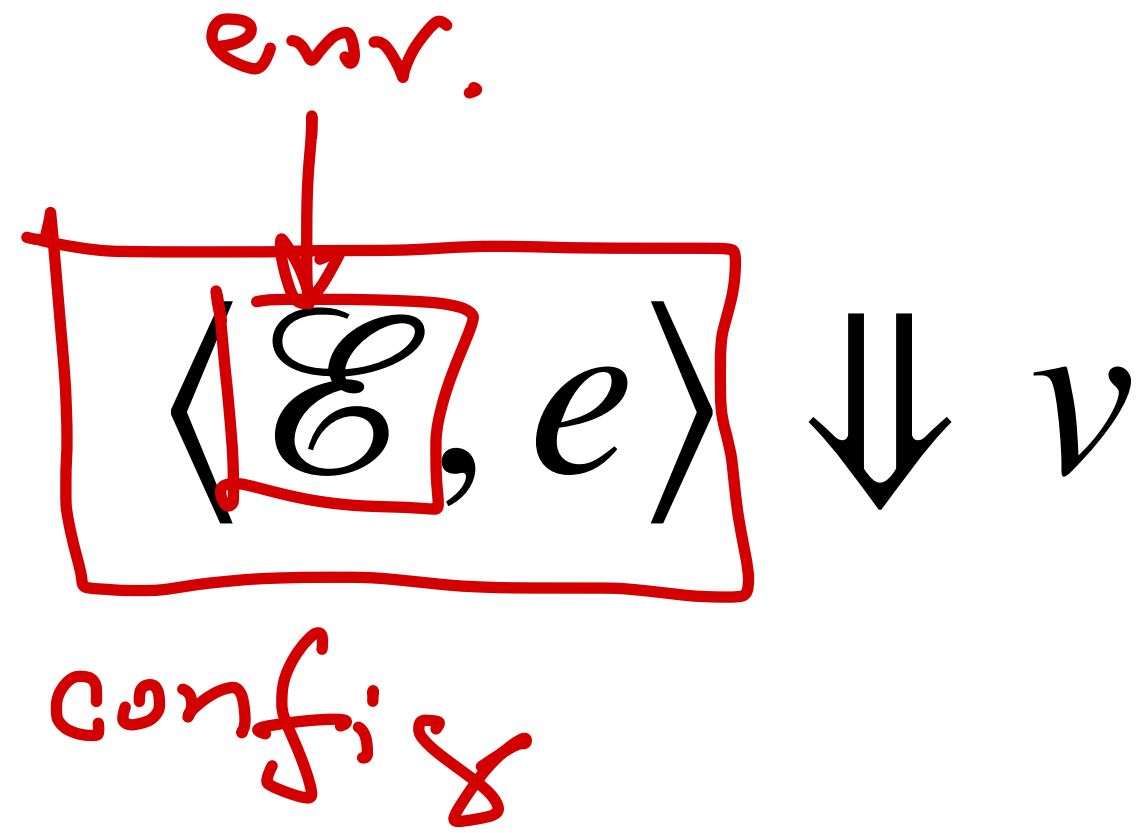
We've already implemented lexical scoping using the substitution model (mini-project 1)

*Why do it again?*

**Answer.** The substitution model is inefficient

Each substitution has to "crawl" through the *entire remainder of the program*

# The Environment Model (High-Level)



Rather than *eagerly* substituting variables, we'll keep track of their values in an environment

We'll then evaluate *relative* to the environment, *lazily* filling in variable values along the way

Now the **configurations** in our semantics have nonempty state

(This might feel a bit more natural than substitution from the perspective of imperative languages)

# Lambda Calculus<sup>+</sup> (Syntax)

$\langle \text{expr} \rangle ::= \lambda \langle \text{var} \rangle . \langle \text{expr} \rangle$   
                  |  $\langle \text{var} \rangle$   
                  |  $\langle \text{expr} \rangle \langle \text{expr} \rangle$   
                  |  $\text{let } \langle \text{var} \rangle = \langle \text{expr} \rangle$   
                  |  $\text{in } \langle \text{expr} \rangle$  } - lets  
                  |  $\langle \text{num} \rangle$  } - numbers

$\langle \text{val} \rangle ::= \lambda \langle \text{var} \rangle . \langle \text{expr} \rangle$   
                  |  $\langle \text{num} \rangle$

This is a grammar for the lambda calculus with let-expressions and numbers

**Challenge Problem.** Rewrite this grammar so that it is not ambiguous

let  $x = 0$  in

let  $f = \lambda y. x$  in

let  $x = 1$  in

$f$  10



# Lambda Calculus<sup>+</sup> (Semantics)

$$\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \lambda x. e} \quad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n} \quad \left. \vphantom{\frac{}{\langle \mathcal{E}, \lambda x. e \rangle \Downarrow \lambda x. e}} \right\} \text{value eval to value.}$$

$$\frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)} \quad \left. \vphantom{\frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}} \right\} \text{var. eval. to it's in value in } \mathcal{E}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x. e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \boxed{\langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v} \quad \left. \vphantom{\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x. e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \boxed{\langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}} \right\} \begin{array}{l} [v_2 / x]e \text{ becomes} \\ \text{essential the} \\ \text{same} \end{array}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \boxed{\langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

**Important. These rules are incorrect!**

# What went wrong?

let  $\boxed{x} = 0$  in  
let  $f = \lambda y. \boxed{x}$  in  
let  $x = 1$  in  $\underbrace{\hspace{1cm}}$  shadow  
 $f\ 0$

*What is the value of this expression?*

We'll see on the next slide that we accidentally implemented dynamic scoping

# The Derivation

$$\frac{}{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow \lambda x . e} \qquad \frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$
$$\frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$
$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x . e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$
$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow v_1 \quad \langle \mathcal{E}[x \mapsto v_1], e_2 \rangle \Downarrow v_2}{\langle \mathcal{E}, \text{let } x = e_1 \text{ in } e_2 \rangle \Downarrow v_2}$$

$$\langle \quad \emptyset \quad , \quad \text{let } x = 0 \text{ in let } f = \lambda y . x \text{ in let } x = 1 \text{ in } f \ 0 \quad \rangle \Downarrow 1$$

# Closures (Looking Ahead)

let  $x = 0$  in  
let  $f = \lambda y. x$  in  
let  $x = 1$  in  
 $f(10)$

*What do we do about this?*

The problem was that  $f$  saw a different value of  $x$  than what it was when  $f$  was evaluated

So functions will need to remember the environments they were defined in

# Summary

To deal with variables in an imperative setting, we need environments

To deal with variables in a function setting, we don't *need* environments, but they will make our implementation more efficient

Dynamic scoping says a variable binding is determined by the computational context whereas lexical scoping says its determine by its lexical (textual) context