

Variables and Environments

Concepts of Programming Languages
Lecture 15

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**

Recap

Recall: Lambda Calculus

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

syntax

Recall: Lambda Calculus

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

syntax

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$

$$\frac{}{(\lambda x . e)(\lambda y . e') \longrightarrow [(\lambda y . e')/x]e}$$

small-step call-by-value

Recall: Lambda Calculus

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

syntax

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$
$$\frac{}{(\lambda x . e)(\lambda y . e') \longrightarrow [(\lambda y . e')/x]e}$$

small-step call-by-value

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{}{(\lambda x . e) e' \longrightarrow [e'/x]e}$$

small-step call-by-name

Recall: Lambda Calculus

::=	$\lambda <var> . <expr>$
	$<var>$
	$<expr> <expr>$

syntax

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$

$$(\lambda x . e) (\lambda y . e') \longrightarrow [(\lambda y . e') / x] e$$

small-step call-by-value

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{}{(\lambda x . e) e' \longrightarrow [e' / x] e}$$

small-step call-by-name

$$\frac{\overline{\lambda x . e \Downarrow \lambda x . e} \quad e_1 \Downarrow \lambda x . e \quad e_2 \Downarrow v_2 \quad [v_2 / x] e \Downarrow v}{e_1 e_2 \Downarrow v}$$

big-step call-by-value

Recall: Lambda Calculus

::=	$\lambda <var> . <expr>$
	$<var>$
	$<expr> <expr>$

syntax

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$

$$\frac{}{(\lambda x . e)(\lambda y . e') \longrightarrow [(\lambda y . e')/x]e}$$

small-step call-by-value

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad \frac{}{(\lambda x . e) e' \longrightarrow [e'/x]e}$$

small-step call-by-name

$$\frac{\overline{\lambda x . e \Downarrow \lambda x . e} \quad e_1 \Downarrow \lambda x . e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

big-step call-by-value

$$\frac{\overline{\lambda x . e \Downarrow \lambda x . e} \quad e_1 \Downarrow \lambda x . e \quad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

big-step call-by-name

Recall: Benefits of CBV

$$(\lambda x . x + x + x + x)e$$

Recall: Benefits of CBV

$$(\lambda x . x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once*

Recall: Benefits of CBV

$$(\lambda x . x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once*

This is good if the variable appears several times in the body of our function

Recall: Benefits of CBV

$$(\lambda x . x + x + x + x)e$$

If we compute the value of an argument before substituting it into the expression, we only have to compute the expression *once*

This is good if the variable appears several times in the body of our function

This is also called **eager**, or **applicative**, or **strict** evaluation (and is what OCaml does)

Recall: Benefits of CBN

$$(\lambda x . \lambda y . x) e_1 e_2$$

Recall: Benefits of CBN

$$(\lambda x . \lambda y . x) e_1 e_2$$

If a variables doesn't appear in our function, then the argument is *not evaluated at all*

Recall: Benefits of CBN

$$(\lambda x . \lambda y . x) e_1 e_2$$

If a variables doesn't appear in our function, then the argument is *not evaluated at all*

Or if an **argument is only seldomly used**, it will only be computed when it is used (e.g, if its computed in a branch of an if-expression that is almost never reached)

Recall: Substitution

$$[y/x](\lambda x . y)$$

Recall: Substitution

$$[y/x](\lambda x . y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Recall: Substitution

$$[y/x](\lambda x . y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Informally. *Replace every instance of x with v*

Recall: Substitution

$$[y/x](\lambda x . y)$$

We write $[v/x]e$ to mean e with v substituted in for x

Informally. *Replace every instance of x with v*

Already things start to break down with this informal definition, e.g., consider the above substitution...

Recall: α -Equivalence

let x = 2 in x + 1

$=_{\alpha}$

let z = 2 in z + 1

OCaml

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

λ -calculus

Recall: α -Equivalence

`let x = 2 in x + 1`

$=_{\alpha}$

`let z = 2 in z + 1`

OCaml

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

λ -calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way (they are **α -equivalent**)

Recall: α -Equivalence

let $x = 2$ in $x + 1$

$=_{\alpha}$

let $z = 2$ in $z + 1$

OCaml

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

λ -calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way (they are **α -equivalent**)

We say that a variable x is **bound** in an expression if it appears in the expression as $(\dots \lambda x . e \dots)$

Recall: α -Equivalence

```
let x = 2 in x + 1
```

 $=_{\alpha}$

```
let z = 2 in z + 1
```

OCaml

$$e_1 =_{\alpha} e_2$$
$$\Downarrow$$

$$[v/x]e_1 =_{\alpha} [v/x]e_2$$

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

λ -calculus

The **principle of name irrelevance** says that any two programs that are the same up to "renaming of variables" should behave exactly the same way (they are **α -equivalent**)

We say that a variable x is **bound** in an expression if it appears in the expression as $(\dots \lambda x. e \dots)$

Substitution should preserve this

Recall: "Correct" Definition of Substitution

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

$$[v/y](\lambda x. e) = \begin{cases} \lambda x. e & \\ \lambda z. [v/y][z/x]e & \\ \lambda x. [v/y]e & \end{cases}$$

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

$[y/x](\lambda y. xz)$
 ~~$\lambda y. yz$~~
 $x = y$
 $x \in FV(v), z \notin FV(e)$
 else
 $[y/x](\lambda z. xz)$
 $[y/x](\lambda w. xz) = \lambda w. yz$

The only problem with this definition is that it now poses an implementation issue

How do we come up with z ?

Well-Scopedness and Closedness

open

$$\lambda x . y$$

closed

$$\lambda x . \lambda y . y$$

Well-Scopedness and Closedness

open
 $\lambda x . y$

closed
 $\lambda x . \lambda y . y$

Definition. (*informal*) An expression e is **well-scoped** if every free variable in e is "in scope" (more on that on later)

Well-Scopedness and Closedness

open
 $\lambda x . y$

closed
 $\lambda x . \lambda y . y$

Definition. (*informal*) An expression e is **well-scoped** if every free variable in e is "in scope" (more on that on later)

Definition. An expression e is **closed** if it has no free variables

Well-Scopedness and Closedness

open

$$\lambda x. y$$

closed

$$\lambda x. \lambda y. y$$

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

Definition. (*informal*) An expression e is **well-scoped** if every free variable in e is "in scope" (more on that on later)

Definition. An expression e is **closed** if it has no free variables

Every closed term is well-scoped

Well-Scopedness Check

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

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

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

Well-Scopedness Check

$$\begin{aligned}[v/y]x &= \begin{cases} v & x = y \\ x & \text{else} \end{cases} \\ [v/y](\lambda x . e) &= \begin{cases} \lambda x . e & x = y \\ \lambda z . [v/y][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x . [v/y]e & \text{else} \end{cases} \\ [v/y](e_1 e_2) &= ([v/y]e_1)([v/y]e_2)\end{aligned}$$

If we only work with closed (well-scoped) expressions, then we don't need to worry about captured variables. **The condition requiring α -renaming never holds!**

Well-Scopedness Check

$$\begin{aligned}[v/y]x &= \begin{cases} v & x = y \\ x & \text{else} \end{cases} \\ [v/y](\lambda x . e) &= \begin{cases} \lambda x . e & x = y \\ \lambda z . [v/y][z/x]e & x \in FV(v), z \notin FV(e) \\ \lambda x . [v/y]e & \text{else} \end{cases} \\ [v/y](e_1 e_2) &= ([v/y]e_1)([v/y]e_2)\end{aligned}$$

If we only work with closed (well-scoped) expressions, then we don't need to worry about captured variables. **The condition requiring α -renaming never holds!**

The Takeaway: In mini-project 1, you should check if the expression has a free variable *before* you evaluate it

Practice Problem

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

Give a derivation of the above judgment in both versions of the big-step semantics

$$\frac{\frac{\overline{\lambda x . e \Downarrow \lambda x . e} \quad e_1 \Downarrow \lambda x . e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}}{\lambda x . e \Downarrow \lambda x . e}$$

$$\frac{\frac{\overline{\lambda x . e \Downarrow \lambda x . e} \quad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}}{e_1 \Downarrow \lambda x . e}$$

Answer

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

demo

(lambda calculus)

Variables

Two Major Concerns

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?

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?

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:

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

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

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

Mutability

```
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)

Mutability

```
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

Mutability

```
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:

Mutability

```
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

Mutability

```
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

Scope

Scope

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

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

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:

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

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)

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 mean:

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 mean:

- » the scope of a variable

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 mean:

- » the scope of a variable
- » the scope of a binding

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 mean:

- » the scope of a variable
- » the scope of a binding
- » the scope of a function

Dynamic Scoping

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

Bash

Dynamic Scoping

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

Bash

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

Dynamic Scoping

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

Bash

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

```
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 Scoping

```
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

Lexical Scoping

```
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:

Lexical Scoping

```
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 it's own scope (**let-bindings**)

Lexical Scoping

```
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 it's own scope (**let-bindings**)
- » A block defines the scope of a variable (**python functions**)

Environments

High-Level

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

High-Level

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

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

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**

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 (more on this in mini-project 2)

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 (more on this in mini-project 2)

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

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

Operations

\emptyset

Math

empty

OCaml

$\{x \mapsto 2, y \mapsto 3\} [z \mapsto 5]$

"

\mathcal{E}

env

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

add x v env

$\{x \mapsto 2, y \mapsto 3, z \mapsto 5\}$

$\mathcal{E}(x)$

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

Operations

Math

\mathcal{E}

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

$\mathcal{E}(x)$

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

OCaml

env

add x v env

find_opt x env

find_opt x env = None

Shadowing

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

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

Important: Adding mappings shadows existing mappings!

Dynamic Scoping

Toy Language (Syntax)

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

examples

$x = 1 ;$

$y = \$x ;$

$f() \{ y = 2 \} ;$
 $f ;$

This is a small grammar for a language with, numbers, subroutines, and variable assignments (like Bash)

Toy Language (Semantics)

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

Toy Language (Semantics)

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

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

Toy Language (Semantics)

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

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

$$\mathcal{E}(f) = P \in \mathbb{F}$$

$$\langle \mathcal{E}, f; Q \rangle \longrightarrow \langle \mathcal{E}, P Q \rangle \quad (\text{call})$$

config. ← State (env) ← program

Toy Language (Semantics)

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

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

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

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

Toy Language (Semantics)

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

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

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

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

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

Toy Language (Semantics)

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

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

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

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

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

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

Toy Language (Example)

\emptyset ,

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

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

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

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

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

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

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

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

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

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

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

The Takeaway

The Takeaway

Defining and implementing dynamic scoping is very easy

The Takeaway

Defining and implementing dynamic scoping is very easy

It's like maintaining only a global scope

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

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 ...
```

Didn't we do this?

`let x = v in ...`

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

Didn't we do this?

let x = v in ...

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

Why do it again?

Didn't we do this?

`let x = v in ...`

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

Why do it again?

Answer. The substitution model is inefficient

Didn't we do this?

let x = v in ...

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)

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

The Environment Model (High-Level)

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

The Environment Model (High-Level)

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

And evaluate *relative* to the environment, *lazily* filling in variable values along the way

The Environment Model (High-Level)

$$\langle \mathcal{E}, e \rangle \Downarrow v$$

Idea. We keep track of their values in an *environment*

And evaluate *relative* to the environment, *lazily* filling in variable values along the way

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

Lambda Calculus⁺ (Syntax)

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

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

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

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

$$\frac{}{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow \lambda x . e}$$

$$\frac{}{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\frac{}{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

$$\overline{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow \lambda x . e}$$

$$\overline{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\overline{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x . e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

Lambda Calculus⁺ (Semantics)

Important. These rules are incorrect!

$$\overline{\langle \mathcal{E}, \lambda x . e \rangle \Downarrow \lambda x . e}$$

$$\overline{\langle \mathcal{E}, n \rangle \Downarrow n}$$

$$\overline{\langle \mathcal{E}, x \rangle \Downarrow \mathcal{E}(x)}$$

$$\frac{\langle \mathcal{E}, e_1 \rangle \Downarrow \lambda x . e \quad \langle \mathcal{E}, e_2 \rangle \Downarrow v_2 \quad \langle \mathcal{E}[x \mapsto v_2], e \rangle \Downarrow v}{\langle \mathcal{E}, e_1 e_2 \rangle \Downarrow v}$$

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

Why are these rules incorrect?

let $x = 0$ in

let $f = \lambda y . x$ in

let $x = 1$ in

$f\ 0$

Why are these rules incorrect?

let $x = 0$ in

let $f = \lambda y . x$ in

let $x = 1$ in

$f\ 0$

What is the value of this expression?

Why are these rules incorrect?

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

What is the value of this expression?

We'll see next time that we've *actually implemented dynamic scoping*

Summary

The **scoping** paradigm of a PL determines when/where variable bindings are available

Dynamic scoping is easier to implement, to less user friendly

We use **environments** to maintain variable bindings