Administrivia

Project 1 is due 4/15 by 11:59PM

Minor project errata: **lookup_env** should be **fetch_env.** Please use the latter, that is the one we will be testing

We will hold office hours during sections this week (excluding A6 and A4 will be held in CDS 907).

Variables: Binding and Scoping

Principles of Programming Languages
Lecture 20

CAS CS 320

Objectives

Discuss the semantics of **variable binding.**

Try to understand **scope**, particularly **lexical** vs. **dynamic** scoping.

Keywords

`environment`

`scope`

`mutability`

`assignment`

`dynamic scope`

`lexical scope`

`let-bindings`

`binding-defined scope`

`block-defined scope`

Two Main Concerns about Variables

Are variables **mutable**? Can we change their value? Are there restrictions when we can change a value?

How are variables **scoped**? Dynamically or lexically? Does a binding define its own scope?

Mutability

Mutability (High Level)

```ocaml
let x =
  let y = 2 in
  let y = 3 in
  y
```

OCaml (y is *shadowed*)

```python
y = 2
y = 3
x = y
```

Python (y is *reassigned*)

Mutability (High Level)

```ocaml
let x =
  let y = 2 in
  let y = 3 in
  y
```

OCaml (y is *shadowed*)

```python
y = 2
y = 3
x = y
```

Python (y is *reassigned*)

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

Mutability (High Level)

```ocaml
let x =
  let y = 2 in
  let y = 3 in
  y
```

OCaml (y is *shadowed*)

```python
y = 2
y = 3
x = y
```

Python (y is *reassigned*)

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

We think of variables as

» names when they are immutable

Mutability (High Level)

```
let x =
  let y = 2 in
  let y = 3 in
  y
```

OCaml (y is *shadowed*)

```
y = 2
y = 3
x = y
```

Python (y is *reassigned*)

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

We think of variables as

» names when they are immutable

» (abstract) memory locations when they are mutable

Examples of Immutable Variables

```swift
func g() -> Int {
  let x = 100
  // let x = 200
  return x
}

var y = g()
```

Constants in Swift

```ocaml
let y =
  (* let x = 100 in *)
  let x = 200 in
  x
```

Let-Bindings in OCaml

Examples of Immutable Variables

```swift
func g() -> Int {
  let x = 100
  // let x = 200
  return x
}

var y = g()
```

Constants in Swift

```ocaml
let y =
  (* let x = 100 in *)
  let x = 200 in
  x
```

Let-Bindings in OCaml

We can think of constants in Swift or OCaml as variables that we cannot reassign.

Examples of Immutable Variables

```swift
func g() -> Int {
  let x = 100
  // let x = 200
  return x
}

var y = g()
```

Constants in Swift

```ocaml
let y =
  (* let x = 100 in *)
  let x = 200 in
  x
```

Let-Bindings in OCaml

We can think of constants in Swift or OCaml as variables that we cannot reassign.

Note. In each case we *technically* can create the new binding, but it would shadow the original, and the static analyzer would complain.

Examples of Mutable Variables

```python
def f():
    y = 0
    y += 1
    return y

x = f()
```

Variables in Python

```ocaml
let x =
  let y = ref 0 in
  y := !y + 1; !y
```

References in OCaml

Examples of Mutable Variables

```python
def f():
    y = 0
    y += 1
    return y

x = f()
```

Variables in Python

```ocaml
let x =
  let y = ref 0 in
  y := !y + 1; !y
```

References in OCaml

We think of variables in Python as references to an abstract memory space. We can make these references in OCaml too.

Examples of Mutable Variables

```python
def f():
    y = 0
    y += 1
    return y

x = f()
```

Variables in Python

```ocaml
let x =
  let y = ref 0 in
  y := !y + 1; !y
```

References in OCaml

We think of variables in Python as references to an abstract memory space. We can make these references in OCaml too.

In Python fetching and updating are **implicit** in the syntax.

Examples of Mutable Variables

```python
def f():
    y = 0
    y += 1
    return y

x = f()
```

Variables in Python

```ocaml
let x =
  let y = ref 0 in
  y := !y + 1; !y
```

References in OCaml

We think of variables in Python as references to an abstract memory space. We can make these references in OCaml too.

In Python fetching and updating are **implicit** in the syntax.

In OCaml, fetching (!) and updating (:=) are **explicit**.

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num>
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

Example Program:

```
x = 1;
y = 2;
x = 3;
x;
y;
```

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num>
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

```
Example Program:
x = 1;
y = 2;
x = 3;
x;
y;
```

This is a simple language with assignment statements and a push operation for identifiers.

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num>
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

```
Example Program:
x = 1;
y = 2;
x = 3;
x;
y;
```

This is a simple language with assignment statements and a push operation for identifiers.

We will take a *configuration* to be:

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num>
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

```
Example Program:
x = 1;
y = 2;
x = 3;
x;
y;
```

This is a simple language with assignment statements and a push operation for identifiers.

We will take a *configuration* to be:

environment

(S, E, P) or **ERROR**

stack program

Operational Semantics

Operational Semantics

$$\dfrac{}{\texttt{(S, E , x = n ; P)} \longrightarrow \texttt{(S, }\mathbf{\texttt{update}}\texttt{(E, x, n), P)}}\ \texttt{(assign)}$$

Operational Semantics

$$\frac{}{\texttt{(S, E , x = n ; P)} \longrightarrow \texttt{(S, }\textbf{update}\texttt{(E, x, n), P)}}\ \texttt{(assign)}$$

$$\frac{\textbf{fetch}\texttt{(E, x) = n} \qquad \texttt{n is a number}}{\texttt{(S, E , x ; P)} \longrightarrow \texttt{(n :: S, E, P)}}\ \texttt{(fetch)}$$

Operational Semantics

$$\frac{}{\texttt{(S, E , x = n ; P)} \longrightarrow \texttt{(S, }\textbf{update}\texttt{(E, x, n), P)}}\ \texttt{(assign)}$$

$$\frac{\textbf{fetch}\texttt{(E, x) = n} \qquad \texttt{n is a number}}{\texttt{(S, E , x ; P)} \longrightarrow \texttt{(n :: S, E, P)}}\ \texttt{(fetch)}$$

$$\frac{\textbf{fetch}\texttt{(E, x) is not a number}}{\texttt{(S, E , x ; P)} \longrightarrow \textbf{ERROR}}\ \texttt{(fetchErr)}$$

Operational Semantics

$$\frac{}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{x}\ =\ \mathtt{n}\ ;\ \mathtt{P}) \longrightarrow (\mathtt{S},\ \mathbf{update}(\mathtt{E},\ \mathtt{x},\ \mathtt{n}),\ \mathtt{P})}\ (\mathtt{assign})$$

$$\frac{\mathbf{fetch}(\mathtt{E},\ \mathtt{x})\ =\ \mathtt{n} \qquad \mathtt{n\ is\ a\ number}}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{x}\ ;\ \mathtt{P}) \longrightarrow (\mathtt{n}\ ::\ \mathtt{S},\ \mathtt{E},\ \mathtt{P})}\ (\mathtt{fetch})$$

$$\frac{\mathbf{fetch}(\mathtt{E},\ \mathtt{x})\ \mathtt{is\ not\ a\ number}}{(\mathtt{S},\ \mathtt{E}\ ,\ \mathtt{x}\ ;\ \mathtt{P}) \longrightarrow \mathbf{ERROR}}\ (\mathtt{fetchErr})$$

Question. *What should the stack be after evaluating the program*

```
x = 1; y = 2; x = 3; x; y;
```

Questions

Questions

`What is an environment?`

Questions

What is an environment?

How should updating work if our variables are immutable?

Questions

What is an environment?

How should updating work if our variables are immutable?

How should it work if they are mutable?

Questions

What is an environment?

How should updating work if our variables are immutable?

How should it work if they are mutable?

Does it matter for this toy-language?

Environments

$$(\mathtt{x} \mapsto \mathtt{v} \ , \ \mathtt{y} \mapsto \mathtt{w} \ , \ \mathtt{z} \mapsto \mathtt{n} \ )$$

Environments

$$(x \mapsto v , y \mapsto w , z \mapsto n)$$

An environment is a collection of **bindings.**

Environments

$$(x \mapsto v , \overbrace{y \mapsto w}^{\texttt{binding}} , \underbrace{z}_{\texttt{variable}} \mapsto \underbrace{n}_{\texttt{value}})$$

An environment is a collection of **bindings.**

Environments

$$(x \mapsto v , \overbrace{y \mapsto w}^{\text{binding}} , \underbrace{z}_{\text{variable}} \mapsto \underbrace{n}_{\text{value}})$$

An environment is a collection of **bindings.**

The exact way you implement an environment depends on the situation.

Environments

$$(x \mapsto v \,,\, \overbrace{y \mapsto w}^{\texttt{binding}} \,,\, \underbrace{z}_{\texttt{variable}} \mapsto \underbrace{n}_{\texttt{value}})$$

`An environment is a collection of` **`bindings.`**

`The exact way you implement an environment depends on the situation.`

`In the project we use an association list:`

Environments

$$(x \mapsto v \ ,\ \overbrace{y \mapsto w}^{\text{binding}} \ ,\ \underbrace{z}_{\text{variable}} \mapsto \underbrace{n}_{\text{value}})$$

An environment is a collection of **bindings.**

The exact way you implement an environment depends on the situation.

In the project we use an association list:

(ident * value) list

Immutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) z
  = Some 3

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) w
  = None
```

Immutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) z
  = Some 3

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) w
  = None
```

Immutable updates should shadow existing bindings

Immutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) z
  = Some 3

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) w
  = None
```

Immutable updates should shadow existing bindings

One way to do this is to prepend new bindings

Immutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) z
  = Some 3

fetch (z ↦ 3, x ↦ 1 , z ↦ 2) w
  = None
```

Immutable updates should shadow existing bindings

One way to do this is to prepend new bindings

Again, this is not the only way!

Mutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (x ↦ 1 , z ↦ 3) z
  = Some 3

fetch (x ↦ 1 , z ↦ 3) w
  = None
```

Mutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (x ↦ 1 , z ↦ 3) z
  = Some 3

fetch (x ↦ 1 , z ↦ 3) w
  = None
```

Mutable updates should change existing bindings in the environment

Mutable Fetches and Updates

```
update (x ↦ 1 , z ↦ 2) z 3
  = (z ↦ 3, x ↦ 1 , z ↦ 2)

fetch (x ↦ 1 , z ↦ 3) z
  = Some 3

fetch (x ↦ 1 , z ↦ 3) w
  = None
```

Mutable updates should change existing bindings in the environment

In this case, we might think of the environment as a map (i.e., dictionary)

Understanding Check

Evaluate the program

```
x = 1; y = 2; x = 3; x; y;
```

with mutable and immutable updates. How does the environment differ.

Does the value of the stack differ in each case?

Scope

The Takeaway

Without restricting when and how bindings are accessed, it doesn't matter whether variables are mutable or immutable.

For it to matter, we have to restrict the *scope* of the binding.

Scope

Scope

Scope is a one of the most unclear terms in computer science.

Scope

Scope is a one of the most unclear terms in computer science.

<u>Different versions of the term:</u>

Scope

Scope is a one of the most unclear terms in computer science.

Different versions of the term:

» the scope of a variable

Scope

Scope is a one of the most unclear terms in computer science.

Different versions of the term:

» the scope of a variable

» the scope of a binding

Scope

Scope is a one of the most unclear terms in computer science.

Different versions of the term:

» the scope of a variable

» the scope of a binding

» the scope of a function

Scope

Scope is a one of the most unclear terms in computer science.

<u>Different versions of the term:</u>

» the scope of a variable

» the scope of a binding

» the scope of a function

» scopes in general (i.e., global scope)

Scope

Scope is a one of the most unclear terms in computer science.

<u>Different versions of the term:</u>

» the scope of a variable

» the scope of a binding

» the scope of a function

» scopes in general (i.e., global scope)

» "this variable is not in scope..."

Scope

Scope is a one of the most unclear terms in computer science.

Different versions of the term:

» the scope of a variable

» the scope of a binding

» the scope of a function

» scopes in general (i.e., global scope)

» "this variable is not in scope..."

» this function is not in the scope of this variable..."

Scoping Rules

The **scope** of a binding is where or when a binding 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)

Dynamic Scoping

Dynamic Scoping

```bash
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic Scoping

```bash
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

In its simplest form, dynamic scoping uses a global environment and *any* binding may be referred to *anywhere* in the program.

Dynamic Scoping

```bash
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

In its simplest form, dynamic scoping uses a global environment and *any* binding may be referred to *anywhere* in the program.

This is a temporal view, i.e., was there a computation done beforehand which affected the value of a variable?

Dynamic Scoping

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

(Bash)

Dynamic scoping refers to the idea that variables bindings are determined at run time based on the computational context.

In its simplest form, dynamic scoping uses a global environment and *any* binding may be referred to *anywhere* in the program.

This is a temporal view, i.e., was there a computation done beforehand which affected the value of a variable?

(It is uncommon in modern programming languages, but it is typically easier to implement)

Example: Bash

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

Example: Bash

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

What does this program print?

Example: Bash

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

What does this program print?

(**echo** is like print)

Example: Bash

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

What does this program print?

(**echo** is like print)

Answer: 0

Example: Project 1

```
def F  0 |> X  #G  ;
def G  X |> Y      ;
1 |> X  #F  Y .
```

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

Example: Project 1

```
def F  0 |> X  #G  ;
def G  X |> Y      ;
1 |> X  #F  Y .
```

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

What do these programs print?

Example: Project 1

```
def F  0 |> X  #G  ;
def G  X |> Y      ;
1 |> X  #F  Y .
```

```
f() { x=0; g; }
g() { y=$x; }
x=1; f; echo $y;
```

What do these programs print?

Answer: Both 0

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

```
Example Program:
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

Example Program:

```
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

This is a simple language with assignment statements and a push operation for identifiers **and subroutines.**

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

```
Example Program:
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

This is a simple language with assignment statements and a push operation for identifiers **and subroutines.**

We will take a *configuration* to be:

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

Example Program:

```
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

This is a simple language with assignment statements and a push operation for identifiers **and subroutines.**

We will take a *configuration* to be:

environment

(S, E, P) or **ERROR**

stack program

Operational Semantics

$$\frac{}{\texttt{(S, E , x = n ; P)} \longrightarrow \texttt{(S, }\mathbf{update}\texttt{(E, x, n), P)}}\ \texttt{(assign)}$$

$$\frac{\mathbf{fetch}\texttt{(E, x) = n} \qquad \texttt{n is a number}}{\texttt{(S, E , x ; P)} \longrightarrow \texttt{(n :: S, E, P)}}\ \texttt{(fetchPush)}$$

$$\frac{\mathbf{fetch}\texttt{(E, f) = Q} \qquad \texttt{Q is a program}}{\texttt{(S, E , f ; P)} \longrightarrow \texttt{(S, E, Q P)}}\ \texttt{(fetchCall)}$$

$$\frac{\mathbf{fetch}\texttt{(E, x) is None}}{\texttt{(S, E , x ; P)} \longrightarrow \mathbf{ERROR}}\ \texttt{(fetchErr)}$$

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

```
([],  [],   f = { x = 0; g; }; g = { x; }; x = 1; f;) ⟶
```

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

`([],  [],   f = { x = 0; g; }; g = { x; }; x = 1; f;)` $\longrightarrow$

`([],  [("f" ↦ x = 0; g;)],   g = { x; }; x = 1; f;)` $\longrightarrow$

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

```
([], [], f = { x = 0; g; }; g = { x; }; x = 1; f;) ⟶
([], [("f" ↦ x = 0; g;)], g = { x; }; x = 1; f;) ⟶
([], [("g" ↦ x;) ("f" ↦ x = 0; g;)], x = 1; f;) ⟶
```

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

```
([],  [],    f = { x = 0; g; }; g = { x; }; x = 1; f;) ⟶

([],  [("f" ↦ x = 0; g;)],    g = { x; }; x = 1; f;) ⟶

([],  [("g" ↦ x;)  ("f" ↦ x = 0; g;)],    x = 1; f;) ⟶

([],  [("x" ↦ 1)   ("g" ↦ x;)  ("f" ↦ x = 0; g;)],    f;) ⟶
```

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

$$([\,],\ [\,],\ \texttt{f = \{ x = 0; g; \}; g = \{ x; \}; x = 1; f;}) \longrightarrow$$

$$([\,],\ [(\texttt{"f"} \mapsto \texttt{x = 0; g;})],\ \texttt{g = \{ x; \}; x = 1; f;}) \longrightarrow$$

$$([\,],\ [(\texttt{"g"} \mapsto \texttt{x;})\ (\texttt{"f"} \mapsto \texttt{x = 0; g;})],\ \texttt{x = 1; f;}) \longrightarrow$$

$$([\,],\ [(\texttt{"x"} \mapsto \texttt{1})\ (\texttt{"g"} \mapsto \texttt{x;})\ (\texttt{"f"} \mapsto \texttt{x = 0; g;})],\ \texttt{f;}) \longrightarrow$$

$$([\,],\ [(\texttt{"x"} \mapsto \texttt{1})\ (\texttt{"g"} \mapsto \texttt{x;})\ (\texttt{"f"} \mapsto \texttt{x = 0; g;})],\ \texttt{x = 0; g;}) \longrightarrow$$

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

```
([],  [],              f = { x = 0; g; }; g = { x; }; x = 1; f;) ⟶
([],  [("f" ↦ x = 0; g;)],     g = { x; }; x = 1; f;) ⟶
([],  [("g" ↦ x;)  ("f" ↦ x = 0; g;)],    x = 1; f;) ⟶
([],  [("x" ↦ 1)   ("g" ↦ x;)  ("f" ↦ x = 0; g;)],   f;) ⟶
([],  [("x" ↦ 1)   ("g" ↦ x;)  ("f" ↦ x = 0; g;)],   x = 0; g;) ⟶
([],  [("x" ↦ 0)   ("g" ↦ x;)  ("f" ↦ x = 0; g;)],   g;) ⟶
```

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

$$
\begin{array}{l}
(\texttt{[]},\ \texttt{[]},\ \texttt{f = \{ x = 0; g; \}; g = \{ x; \}; x = 1; f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{g = \{ x; \}; x = 1; f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{x = 1; f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{1)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{1)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{x = 0; g;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{0)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{g;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{0)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{x = 0;}) \longrightarrow
\end{array}
$$

Example: Toy Language

```
f = { x = 0; g; };
g = { x; };
x = 1; f;
```

$$
\begin{array}{l}
(\texttt{[]},\ \texttt{[]},\ \texttt{f = \{ x = 0; g; \}; g = \{ x; \}; x = 1; f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{g = \{ x; \}; x = 1; f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{x = 1; f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{1)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{f;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{1)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{x = 0; g;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{0)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{g;}) \longrightarrow \\
(\texttt{[]},\ \texttt{[("x"} \mapsto \texttt{0)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \texttt{x = 0;}) \longrightarrow \\
(\texttt{[0]},\ \texttt{[("x"} \mapsto \texttt{0)\ ("g"} \mapsto \texttt{x;)\ ("f"} \mapsto \texttt{x = 0; g;)]},\ \epsilon)
\end{array}
$$

Lexical Scoping

Lexical Scoping

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

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical Scoping

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

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

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

Lexical Scoping

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

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

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

A binding may be referred to within the delimited textual area of the code

Lexical Scoping

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

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

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

A binding may be referred to within the delimited textual area of the code

This is also called static scoping because, in theory, scoping errors can be found before the program is run

Lexical Scoping

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

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

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

A binding may be referred to within the delimited textual area of the code

This is also called static scoping because, in theory, scoping errors can be found before the program is run

(This is far more common in modern programming languages)

Restricting Scope

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

(Python)

```ocaml
let x = 0                     scope of x
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Restricting Scope

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

scope of f

(Python)

```
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

Restricting Scope

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

(Python)

```ocaml
let x = 0                 scope of x
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

» The binding defines its own scope (e.g. let-bindings)

Restricting Scope

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

scope of f

(Python)

```ocaml
let x = 0
let y = let x = 1 in x

let _ = assert(y = 1)
let _ = assert(x = 0)
```

scope of x

(OCaml)

Lexical scoping allows us to restrict the scope of a binding. This tends to happen in two ways:

» The binding defines its own scope (e.g. let-bindings)

» A subroutine or code block defines a scope (e.g. python function)

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

Example Program:

```
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

This is a simple language with assignment statements and a push operation for identifiers **and subroutines.**

We will take a *configuration* to be:

(S, E, P) or **ERROR**

Toy Stack-Oriented Language

```
<prog>  ::= { <com> ; }
<val>   ::= <num> | { <prog> }
<com>   ::= <ident> = <val> | <ident>
<ident> ::= w | x | y | z
<num>   ::= 0 | 1 | 2 | 3 | 4 | 5 | 6
```

Example Program:

```
x = 1;
f = {
  x = 2;
  x;
};
f;
x;
```

This is a simple language with assignment statements and a push operation for identifiers **and subroutines.**

We will take a *configuration* to be:

environment

(S, E, P) or **ERROR**

stack program

Example

```
x = 1;
f = { x; }
x = 2;
f ;
```

What should be at the top of the stack after evaluation?

Dynamic scoping: 2

Lexical Scoping: 1? 2? Depends. Is x mutable? What is the scope of x after assignment?

(we'll come back to this next week)

An Overview

```
Dynamically scoped:
  Bash variables

Lexically scoped, immutable, binding defined:
  OCaml let-bindings

Lexically scoped, mutable, block defined:
  Local variables in Python functions

Lexically scoped, mutable, binding defined:
  OCaml references
```

demo

(epilogue: OCaml References)