

# **The Substitution Model**

## **Concepts of Programming Languages**

# Outline

- » Discuss **substitution** and the pitfalls to avoid
- » Demo an **implementation** of the lambda calculus
- » *If we have time:* Discuss the difference between **lexical** and **dynamic** scoping

# Recap

# Recall: Lambda Calculus

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

syntax

# Recall: Lambda Calculus

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

small-step call-by-value

fun  $x \rightarrow e$

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

  |  $\langle \text{var} \rangle$

  |  $\langle \text{expr} \rangle \langle \text{expr} \rangle$

$(\text{fun } x \rightarrow x)^2$  syntax

# Recall: Lambda Calculus

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

syntax

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

small-step call-by-value

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

small-step call-by-name

# Recall: Lambda Calculus

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

syntax

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

small-step call-by-value

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

small-step call-by-name

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

big-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**

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

**small-step call-by-value**

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

**small-step call-by-name**

$$\frac{}{\lambda x. e \Downarrow \lambda x. e}$$

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

**big-step call-by-value**

$$\frac{}{\lambda x. e \Downarrow \lambda x. e}$$

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

**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

$$(\lambda x. \lambda y. x)e_1e_2$$

# Recall: Benefits of CBN

$$(\lambda x. \lambda y. x)e_1e_2$$

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

# Recall: Benefits of CBN

$$(\lambda x. \lambda y. x)e_1e_2$$

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)

# 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

# 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

$$\frac{\frac{\frac{\lambda x. e \Downarrow \lambda x. e}{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}}{e_1 e_2 \Downarrow v}}{\text{big-step call-by-value}}$$

# 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

$$\frac{\frac{\frac{\lambda x. e \Downarrow \lambda x. e}{e_1 \Downarrow \lambda x. e} \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}}{\text{big-step call-by-value}}$$

$$\frac{\frac{\frac{\lambda x. e \Downarrow \lambda x. e}{e_1 \Downarrow \lambda x. e} \quad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}}{\text{big-step call-by-name}}$$

# Answer

$$\frac{}{\lambda x. e \Downarrow \lambda x. e} \quad (1)$$

$$\frac{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v} \quad (2)$$

$$\frac{}{\lambda x. e \Downarrow \lambda x. e}$$

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

$$[v_2/z]z = [\lambda q. q/z]z = \lambda q. q$$

$$[\lambda q. q/x] \lambda y. y = \lambda y. y$$

$$[v_2/x]e$$

$$\lambda y. y \Downarrow \lambda y. y$$

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

$$\lambda z. z \Downarrow \lambda z. z$$

$$\lambda q. q \Downarrow \lambda q. q$$

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

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

# Answer

$$\overline{\lambda x. e \Downarrow \lambda x. e}$$

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

$$\overline{\lambda x. e \Downarrow \lambda x. e}$$

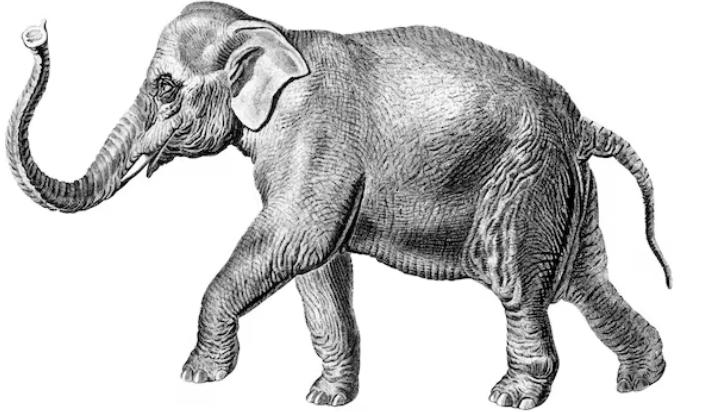
$$\frac{e_1 \Downarrow \lambda x. e \quad [e_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

Exercise :  CBN

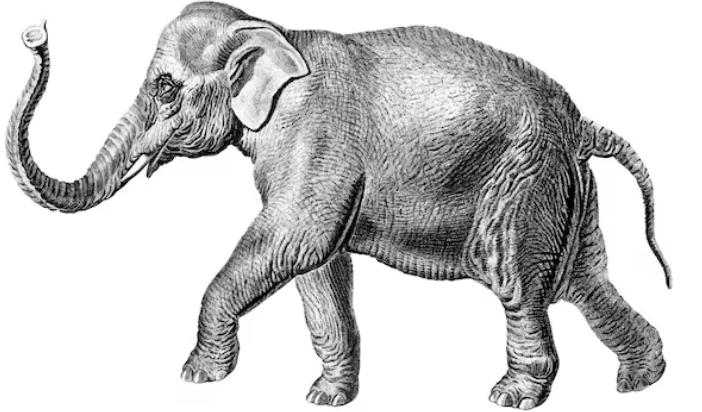
# **Substitution**

# Substitution



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

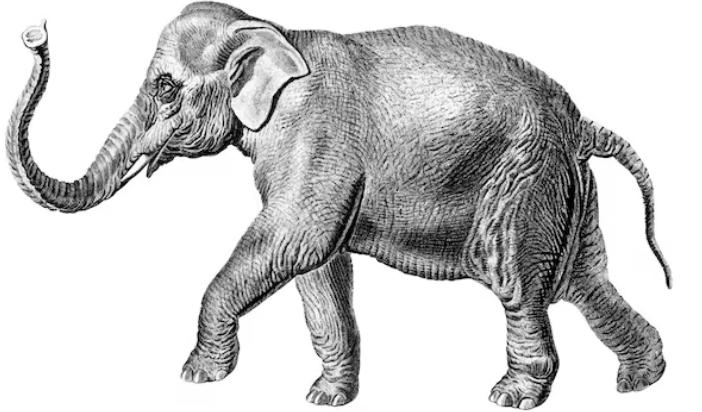
# Substitution



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

It's time to get more formal about **substitution**

# Substitution

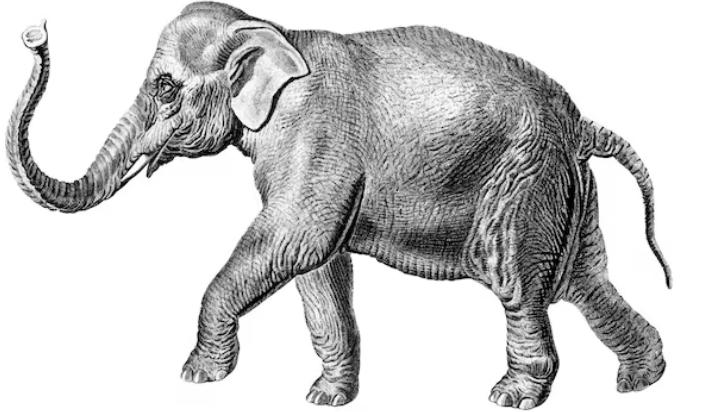


$$\frac{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

It's time to get more formal about **substitution**

We've been able to get by on our intuitions for a while, but our intuitions won't help us *implement* substitution (which is *difficult*)

# Substitution



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

It's time to get more formal about **substitution**

We've been able to get by on our intuitions for a while, but our intuitions won't help us *implement* substitution (which is *difficult*)

*We need to understand why...*

# Recall: Notation

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

# Recall: Notation

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

$\stackrel{[v/x]}{e}$

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

# Recall: Notation

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

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

**Informally.** Replace every instance of  $x$  with  $v$

# Recall: Notation

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

A red circle highlights the  $x$  in  $[y/x]$ . A red arrow points from this circle to the  $x$  in  $(\lambda x . y)$ .

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...

# Our Primary Concern

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

*is not equivalent to*

$$\lambda y . y$$

# Our Primary Concern

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

*is not equivalent to*

$$\lambda y . y$$

However we define substitution shouldn't *change the underlying behavior of a function*

# Our Primary Concern

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

*is not equivalent to*

$$\lambda y . y$$

However we define substitution shouldn't *change the underlying behavior of a function*

**The Key Point:** A function does not depend on our choice of variable names

# $\alpha$ -Equivalence

```
let x = 2 in x + 1  
      = $\alpha$   
let z = 2 in z + 1
```

0Caml

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

$\lambda$ -calculus

# $\alpha$ -Equivalence

```
let x = 2 in x + 1  
      = $\alpha$   
let z = 2 in z + 1
```

0Caml

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

$\lambda$ -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  $\alpha$ -equivalent)

# $\alpha$ -Equivalence

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

 $\equiv_{\alpha}$ 

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

0Caml

 $\lambda x. \lambda y. x \equiv_{\alpha} \lambda v. \lambda w. v$ 

$\lambda$ -calculus

$x \leftrightarrow v$

$y \leftrightarrow w$

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  $\alpha$ -equivalent)

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

# $\alpha$ -Equivalence

```
let x = 2 in x + 1  
      = $\alpha$   
let z = 2 in z + 1
```

0Caml

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

$\lambda$ -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  $\alpha$ -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**

# Preserving $\alpha$ -equivalent

$$[\gamma(x)] \lambda x . y =_{\alpha} [\gamma(x)] \lambda z . y$$

$$\boxed{\lambda y . y} \neq_{\alpha} \lambda z . y$$

What does it mean to *preserve  $\alpha$ -equivalence*?

**The idea:** If two expressions are  $\alpha$ -equivalent, then they should *remain*  $\alpha$ -equivalent after any substitution

# Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \text{else} \end{cases} \quad (1)$$

$$[v/y](\lambda x . e) = \lambda x . [v/y]e \quad (2)$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \quad (3)$$

# Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \text{else} \end{cases} \quad (1)$$

$$[v/y](\lambda x . e) = \lambda x . [v/y]e \quad (2)$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \quad (3)$$

1. Replace every  $y$  with  $v$ , leave other variables

# Definition (First Attempt)

$$[v/y]x = \begin{cases} v & x = y \\ x & \text{else} \end{cases} \quad (1)$$

$$[v/y](\lambda x . e) = \lambda x . [v/y]e \quad (2)$$

$$[v/y](e_1 e_2) = ([v/y]e_1)([v/y]e_2) \quad (3)$$

1. Replace every  $y$  with  $v$ , leave other variables
2. Replace  $y$  with  $v$  in the body of a function

# Definition (First Attempt)

$$[\nu/y]x = \begin{cases} \nu & x = y \\ x & \text{else} \end{cases} \quad (1)$$

$$[\nu/y](\lambda x . e) = \lambda x . [\nu/y]e \quad (2)$$

$$[\nu/y](e_1e_2) = ([\nu/y]e_1)([\nu/y]e_2) \quad (3)$$

1. Replace every  $y$  with  $\nu$ , leave other variables
  2. Replace  $y$  with  $\nu$  in the body of a function
  3. Replace  $y$  with  $\nu$  in both subexpressions of an application
- (This is an example of an *inductive definition*)

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

# Example

$$[\lambda z. z / y]$$

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

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

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

$$\lambda x. (\lambda z. z) \times (\lambda z. z)$$

$$[v/y](\lambda x. e) = \lambda x. [v/y]e$$

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

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

# Problem Case I

$$[v/y](\lambda x. e) = \lambda x. [v/y]e$$

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

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

$$\lambda z. [\gamma /_x]z = \lambda x. [\gamma /_x]x =$$

$$\lambda z. z \neq \lambda x. \gamma$$

We shouldn't be allowed to substitute  $x$  if it's the argument of a function

This may *change the behavior* of a function

# Definition (Second Attempt)

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

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

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

We can handle the problem case directly in our definition. *Check the bound variable before we substitute in the body of a function*

# Definition (Second Attempt)

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

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

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

We can handle the problem case directly in our definition. *Check the bound variable before we substitute in the body of a function*

**Is there still a problem?**

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

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

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

# Problem Case II

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

$$\lambda y . [y/x] x =$$

$$\lambda y . y$$

We're not replacing a bound variable, but we are substituting an expression that has variables which *became* bound

The variable  $y$  is said to be **captured** in this (incorrect) substitution

# Free and Bound Variables

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x . e) = FV(e) \setminus \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

# Free and Bound Variables

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x . e) = FV(e) \setminus \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

Definition. A variable  $x$  is **free** in  $e$  if it does not appear **bound** by a  $\lambda$ .  
Formally:

# Free and Bound Variables

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x . e) = FV(e) \setminus \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

Definition. A variable  $x$  is **free** in  $e$  if it does not appear **bound** by a  $\lambda$ .  
Formally:

1.  $x$  is free in  $x$

# Free and Bound Variables

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x . e) = FV(e) \setminus \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

Definition. A variable  $x$  is **free** in  $e$  if it does not appear **bound** by a  $\lambda$ .  
Formally:

1.  $x$  is free in  $x$
2.  $x$  is free in  $\lambda y . e$  if it is free in  $e$  and  $x \neq y$

# Free and Bound Variables

Sum +  $\rightarrow e$

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x . e) = FV(e) \setminus \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

Definition. A variable  $x$  is **free** in  $e$  if it does not appear **bound** by a  $\lambda$ .  
Formally:

1.  $x$  is free in  $x$
2.  $x$  is free in  $\lambda y . e$  if it is free in  $e$  and  $x \neq y$
3.  $x$  is free in  $e_1 e_2$  if  $x$  is free in  $e_1$  or  $e_2$

# Free and Bound Variables

$$FV(\lambda x. x \cdot y) =$$

$$FV(x) = \{x\} \quad (1)$$

$$FV(\lambda x. e) = FV(e) \setminus \{x\} \quad (2)$$

$$FV(e_1 e_2) = FV(e_1) \cup FV(e_2) \quad (3)$$

$$FV(x \cdot y) \setminus \{x\}$$

$$(FV(x) \cup FV(y)) \setminus \{x\}$$
$$\{x, y\} \setminus \{x\} = \{y\}$$

Definition. A variable  $x$  is **free** in  $e$  if it does not appear **bound** by a  $\lambda$ .

Formally:

1.  $x$  is free in  $x$
2.  $x$  is free in  $\lambda y. e$  if it is free in  $e$  and  $x \neq y$
3.  $x$  is free in  $e_1 e_2$  if  $x$  is free in  $e_1$  or  $e_2$

Definition. A variable  $x$  is **free** in  $e$  if  $x \in FV(e)$  as above

# Definition (Third Attempt)

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

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

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

Since we're interested in  $\alpha$ -equivalence, we can first *replace* the bound variable and *substitute* it in the body of the function. This is called  **$\alpha$ -renaming**

# Definition (Third Attempt)

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

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

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

Since we're interested in  $\alpha$ -equivalence, we can first *replace* the bound variable and *substitute* it in the body of the function. This is called  $\alpha$ -renaming

**Is there still a problem?**

# Problem Case III

$$\begin{aligned}FV(x) &= \{x\} \\FV(\lambda x. e) &= FV(e) \setminus \{x\} \\FV(e_1 e_2) &= FV(e_1) \cup FV(e_2)\end{aligned}$$

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

$$[x/y](\lambda \cancel{x}. \cancel{x}yz) =$$

$$\lambda z. [x/y] zyz =$$

$$\lambda z. z \times z \quad \cancel{\lambda z. z \times z}$$

This isn't exactly a problem, but we *have to be careful about which variable to replace the bound variable  $x$  with*

If we choose  $z$ , then we capture a *different* variable!

# "Correct" Definition

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

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

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

# "Correct" Definition

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

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

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

Finally a definition, that works. Sort of...

# "Correct" Definition

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

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

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

Finally a definition, that works. Sort of...

The only problem with this definition is that it now poses an *implementation issue*. **How do we come up with  $z$ ?**

# Well-Scopedness/Closedness

open

$$\lambda x . y$$

closed

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

# Well-Scopedness/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"

# Well-Scopedness/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"

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

# Well-Scopedness/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"

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

*Closed terms are well-scoped*

# Our Solution: Well-Scopedness Check

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

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

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

no FV at  $\alpha$ )

$x = y$

$x \in FV(v), z$  is fresh

else

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

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

demo  
(lambda calculus)

demo  
(lambda calculus)

# Variable Scoping

# 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

# 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](#))

# Tradeoffs

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

dynamic

vs.

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

lexical

Implementing dynamic scoping is way easier... (we'll see this in lab)

But **every modern programming language** implements lexical scoping

# Looking Ahead: Didn't we do this?

let x = v in ...

# Looking Ahead: 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?*

# Looking Ahead: 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

# Looking Ahead: 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*

# Next Time: The Environment Model

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

# Next Time: The Environment Model

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

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

# Next Time: The Environment Model

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

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

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

# Next Time: The Environment Model

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

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 configurations in our semantics will have nonempty state*

# Summary

**Substitution** is a bit tricky to define correctly but any definition must preserve  $\alpha$ -equivalence

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