

# The Substitution Model

**Concepts of Programming Languages**  
**Lecture 15**

# Outline

Look formally at the **lambda calculus** and its semantics

Discuss **substitution** and the pitfalls to avoid

Demo an **implementation** of the lambda calculus

# Recap

# Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

# Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

# Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

**Notation.** We write  $e \longrightarrow e'$  to mean  $e$  reduces to  $e'$  in a single step

# Recall: Small-Step Semantics

$$(S, p) \longrightarrow (S', p')$$

Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

**Notation.** We write  $e \longrightarrow e'$  to mean  $e$  reduces to  $e'$  in a single step

In general, we define small-step semantics on a **configuration**, which is a program together with some stateful information

# Recall: Small-Step Semantics

$$(S, \overset{\text{program}}{p}) \longrightarrow (S', \overset{\text{transformed program}}{p'})$$

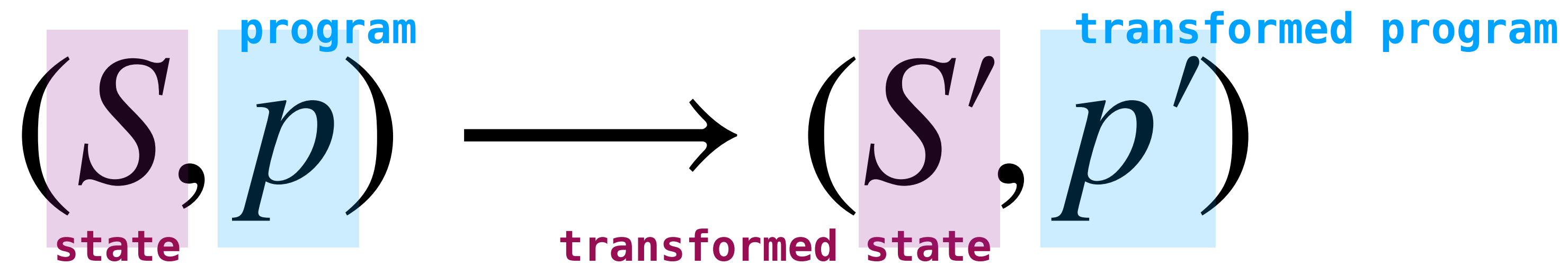
Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

**Notation.** We write  $e \longrightarrow e'$  to mean  $e$  reduces to  $e'$  in a single step

In general, we define small-step semantics on a **configuration**, which is a program together with some stateful information



# Recall: Small-Step Semantics

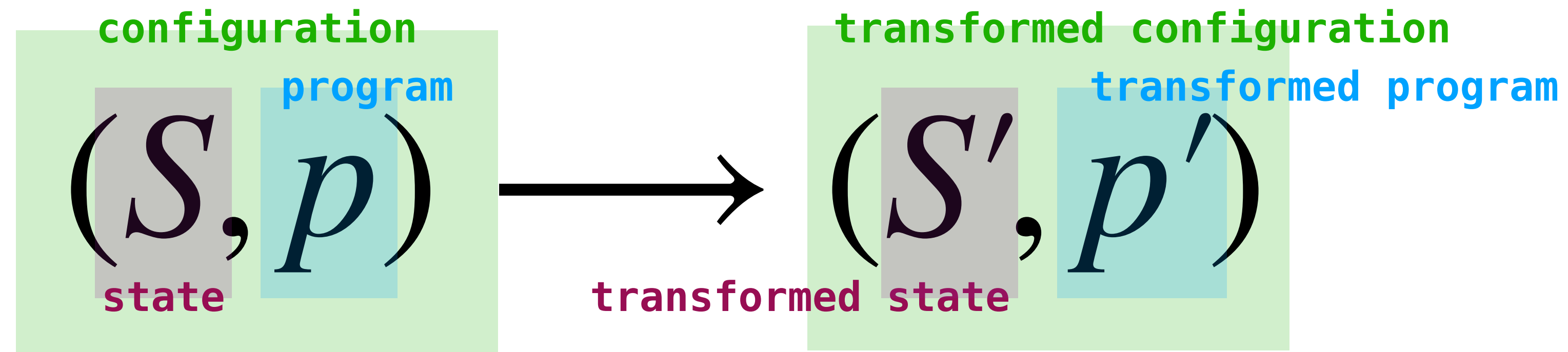


Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

**Notation.** We write  $e \longrightarrow e'$  to mean  $e$  reduces to  $e'$  in a single step

In general, we define small-step semantics on a **configuration**, which is a program together with some stateful information

# Recall: Small-Step Semantics

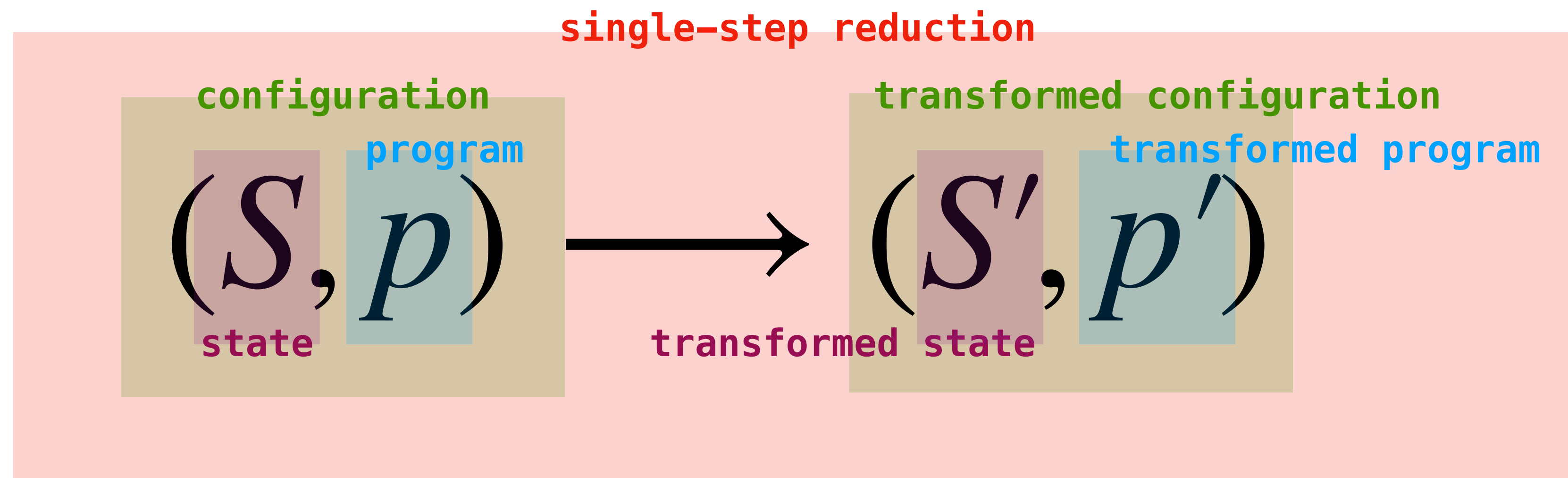


Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

**Notation.** We write  $e \longrightarrow e'$  to mean  $e$  reduces to  $e'$  in a single step

In general, we define small-step semantics on a **configuration**, which is a program together with some stateful information

# Recall: Small-Step Semantics

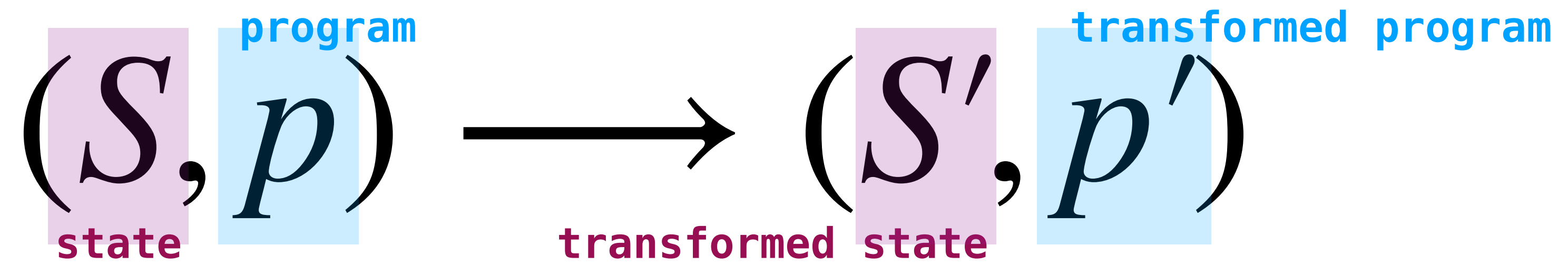


Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

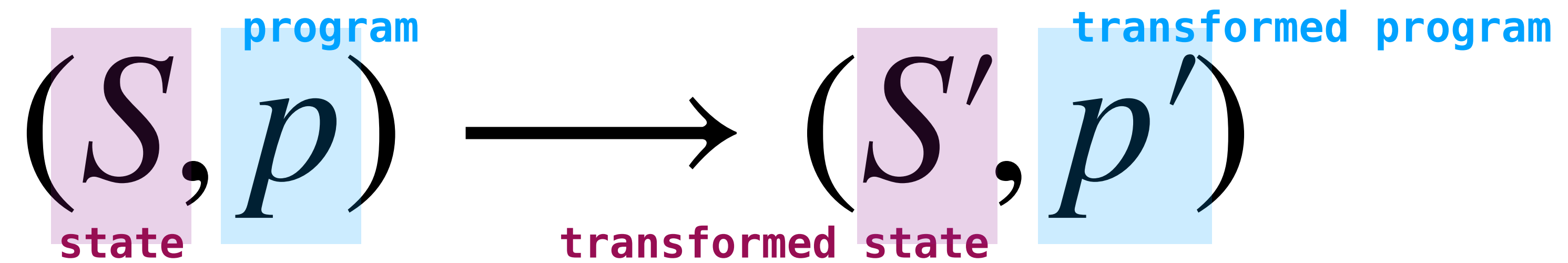
**Notation.** We write  $e \longrightarrow e'$  to mean  $e$  reduces to  $e'$  in a single step

In general, we define small-step semantics on a **configuration**, which is a program together with some stateful information

# Recall: High-Level

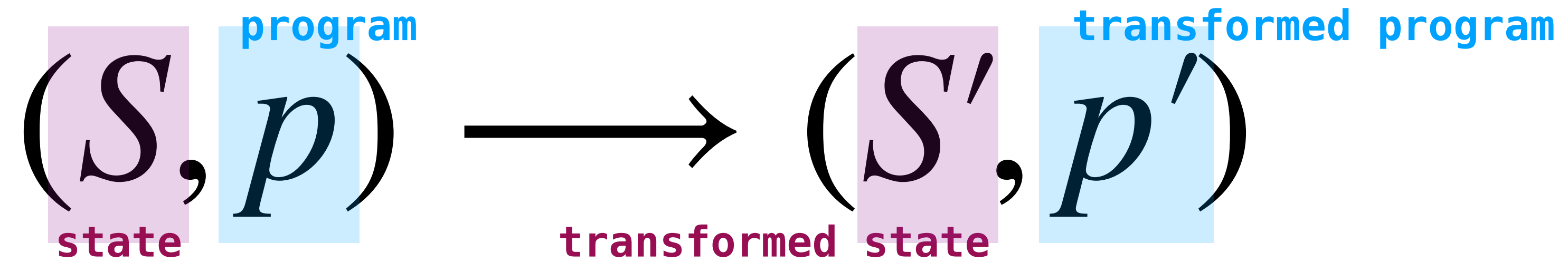


# Recall: High-Level



When we define the small-step semantics of PL, we need to define **three** things:

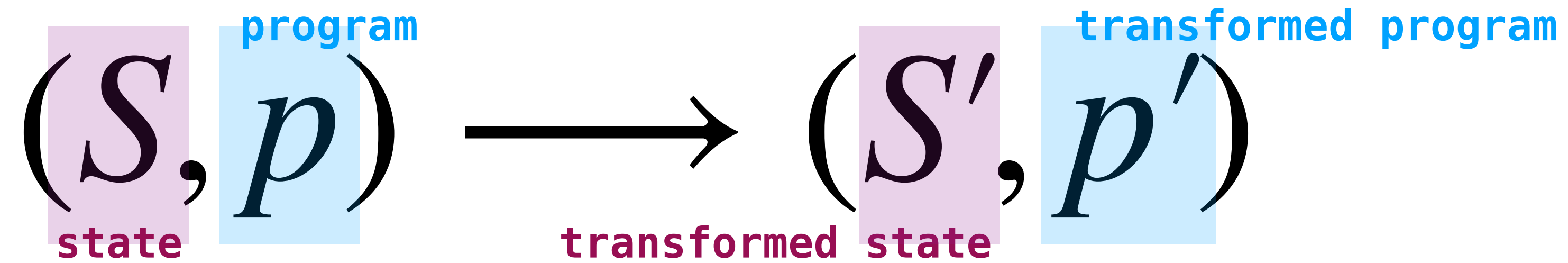
# Recall: High-Level



When we define the small-step semantics of PL, we need to define **three** things:

» What kind of **state** are we manipulating?

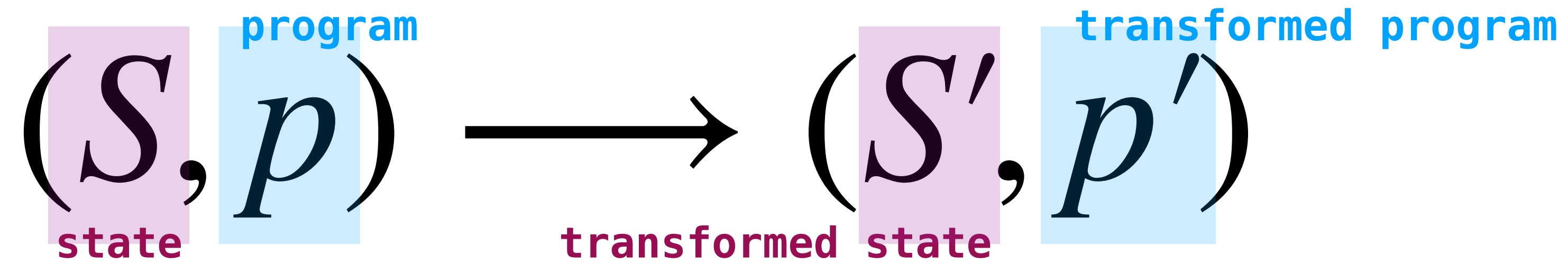
# Recall: High-Level



When we define the small-step semantics of PL, we need to define **three** things:

- » What kind of **state** are we manipulating?
- » What **rules** describe how to transform configurations?

# Recall: High-Level



When we define the small-step semantics of PL, we need to define **three** things:

- » What kind of **state** are we manipulating?
- » What **rules** describe how to transform configurations?
- » What are the **values** of our PL (i.e., when are we done reducing)?



# Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )  
         | <bool> | <int>  
<op>    ::= add | sub | eq  
<bool>  ::= true | false  
<int>   ::= ...
```

# Recall: Example

State:  $\emptyset$

```
<expr> ::= ( <op> <expr> <expr> )  
         | <bool> | <int>  
<op>    ::= add | sub | eq  
<bool>  ::= true | false  
<int>   ::= ...
```

# Recall: Example

State:  $\emptyset$

Rules:

$\langle \text{expr} \rangle$	$::=$	$( \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{expr} \rangle )$
		$  \langle \text{bool} \rangle \quad   \quad \langle \text{int} \rangle$
$\langle \text{op} \rangle$	$::=$	$\text{add} \quad   \quad \text{sub} \quad   \quad \text{eq}$
$\langle \text{bool} \rangle$	$::=$	$\text{true} \quad   \quad \text{false}$
$\langle \text{int} \rangle$	$::=$	$\dots$

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left} \qquad \frac{n \text{ is a number} \quad e_2 \longrightarrow e'_2}{(\text{add } n \ e_2) \longrightarrow (\text{add } n \ e'_2)} \text{ add-right}$$

$$\frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{add } n_1 \ n_2) \longrightarrow n_1 + n_2} \text{ add-ok}$$

$$\frac{e_1 \longrightarrow e'_1}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e'_1 \ e_2)} \text{ sub-left} \qquad \frac{n \text{ is a number} \quad e_2 \longrightarrow e'_2}{(\text{sub } n \ e_2) \longrightarrow (\text{sub } n \ e'_2)} \text{ sub-right}$$

$$\frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{sub } n_1 \ n_2) \longrightarrow n_1 - n_2} \text{ sub-ok}$$

# Recall: Example

State:  $\emptyset$

Rules:

$\langle \text{expr} \rangle$	$::=$	$( \langle \text{op} \rangle \langle \text{expr} \rangle \langle \text{expr} \rangle )$
		$  \langle \text{bool} \rangle \quad   \quad \langle \text{int} \rangle$
$\langle \text{op} \rangle$	$::=$	$\text{add} \quad   \quad \text{sub} \quad   \quad \text{eq}$
$\langle \text{bool} \rangle$	$::=$	$\text{true} \quad   \quad \text{false}$
$\langle \text{int} \rangle$	$::=$	$\dots$

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left} \qquad \frac{n \text{ is a number} \quad e_2 \longrightarrow e'_2}{(\text{add } n \ e_2) \longrightarrow (\text{add } n \ e'_2)} \text{ add-right}$$

$$\frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{add } n_1 \ n_2) \longrightarrow n_1 + n_2} \text{ add-ok}$$

$$\frac{e_1 \longrightarrow e'_1}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e'_1 \ e_2)} \text{ sub-left} \qquad \frac{n \text{ is a number} \quad e_2 \longrightarrow e'_2}{(\text{sub } n \ e_2) \longrightarrow (\text{sub } n \ e'_2)} \text{ sub-right}$$

$$\frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{sub } n_1 \ n_2) \longrightarrow n_1 - n_2} \text{ sub-ok}$$

Values:  $\langle \text{int} \rangle$  (i.e., numbers)

# Recall: Example

```
<expr> ::= ( <op> <expr> <expr> )  
         | <bool> | <int>  
<op>    ::= add | sub | eq  
<bool>  ::= true | false  
<int>   ::= ...
```

$$\frac{n \text{ is a number}}{n \Downarrow n} \text{ numEval}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \text{ is a number} \quad v_2 \text{ is a number}}{(\text{add } e_1 \ e_2) \Downarrow v_1 + v_2} \text{ addEval}$$

$$\frac{e_1 \Downarrow v_1 \quad e_2 \Downarrow v_2 \quad v_1 \text{ is a number} \quad v_2 \text{ is a number}}{(\text{sub } e_1 \ e_2) \Downarrow v_1 - v_2} \text{ subEval}$$

We can also give a big-step semantics to this system

# Practice Problem

<code>&lt;prog&gt;</code>	<code>::= { &lt;stmt&gt; ; }</code>
<code>&lt;stmt&gt;</code>	<code>::= rot90   refX   refY</code>

---

$$(s, \text{rot90}; P) \longrightarrow (s \text{ rotated } 90 \text{ deg. clockwise}, P)$$

---

$$(s, \text{refX}; P) \longrightarrow (s \text{ reflected across x-axis}, P)$$

---

$$(s, \text{refY}; P) \longrightarrow (s \text{ reflected across y-axis}, P)$$

*What does  $(\triangle, \text{rot90}; \text{refY}; \text{rot90}; \text{refX};)$  evaluate to?  
Give a sequence of single step reductions (you do not need to give the full multi-step derivation)*

# Answer

$\langle \text{prog} \rangle ::= \{ \langle \text{stmt} \rangle ; \}$
$\langle \text{stmt} \rangle ::= \text{rot90} \mid \text{refX} \mid \text{refY}$

$(\Delta, \text{rot90}; \text{refY}; \text{rot90}; \text{refX};)$

$(\triangleleft, \text{rot90}; \text{refY}; \text{rot90}; \text{refX};) \rightarrow$

$(\triangleright, \text{refY}; \text{rot90}; \text{refX};) \rightarrow$

$(\triangleleft, \text{rot90}; \text{refX};) \rightarrow$

$(\triangle, \text{refX};) \rightarrow$

$(\nabla, \varepsilon) \checkmark$

$\uparrow$   
empty program

# The Lambda Calculus



# High-Level View

```
(fun x -> x x) (fun x -> x x)
```

lambda term called  $\Omega$

# High-Level View

```
(fun x -> x x) (fun x -> x x)
```

lambda term called  $\Omega$

The **lambda calculus** is the *simplest functional programming language*. It only has:

# High-Level View

```
(fun x -> x x) (fun x -> x x)
```

lambda term called  $\Omega$

The **lambda calculus** is the *simplest functional programming language*. It only has:

» variables

# High-Level View

```
(fun x -> x x) (fun x -> x x)
```

lambda term called  $\Omega$

The **lambda calculus** is the *simplest functional programming language*. It only has:

- » variables
- » anonymous functions

# High-Level View

```
(fun x -> x x) (fun x -> x x)
```

lambda term called  $\Omega$

The **lambda calculus** is the *simplest functional programming language*. It only has:

- » variables
- » anonymous functions
- » function application

# High-Level View

```
(fun x -> x x) (fun x -> x x)
```

lambda term called  $\Omega$

The **lambda calculus** is the *simplest functional programming language*. It only has:

- » variables
- » anonymous functions
- » function application

It's also **untyped**, so *anything* can be applied to *anything*

# demo

(OCaml and Python)

# History





# History

The lambda calculus was introduced by **Alonzo Church** in the 1930s



# History

The lambda calculus was introduced by **Alonzo Church** in the 1930s

Church was trying to give a *foundation of mathematics* (did not succeed) and extracted from that work the lambda calculus



# History

The lambda calculus was introduced by **Alonzo Church** in the 1930s

Church was trying to give a *foundation of mathematics* (did not succeed) and extracted from that work the lambda calculus

The lambda calculus **as powerful** as every model of computation (Turing Machines, Register Machines, etc.)



# Syntax

```
<expr> ::= fun <var> -> <expr>  
         | <var>  
         | <expr> <expr>  
         | ( <expr> )  
<var>   ::= a | b | . . . | y | z
```

# Syntax

<code>&lt;expr&gt;</code>	<code>::=</code>	<code>fun</code>	<code>&lt;var&gt;</code>	<code>-&gt;</code>	<code>&lt;expr&gt;</code>
			<code>&lt;var&gt;</code>		
			<code>&lt;expr&gt;</code>	<code>&lt;expr&gt;</code>	
			<code>(</code>	<code>&lt;expr&gt;</code>	<code>)</code>
<code>&lt;var&gt;</code>	<code>::=</code>	<code>a</code>	<code> </code>	<code>b</code>	<code> </code>
		<code>...</code>	<code> </code>	<code>y</code>	<code> </code>
				<code>z</code>	

*This presentation is technically ambiguous (why?)*

# Syntax

<code>&lt;expr&gt;</code>	<code>::=</code>	<code>fun</code>	<code>&lt;var&gt;</code>	<code>-&gt;</code>	<code>&lt;expr&gt;</code>
			<code>&lt;var&gt;</code>		
			<code>&lt;expr&gt;</code>	<code>&lt;expr&gt;</code>	
			<code>(</code>	<code>&lt;expr&gt;</code>	<code>)</code>
<code>&lt;var&gt;</code>	<code>::=</code>	<code>a</code>	<code> </code>	<code>b</code>	<code> </code>
			<code>...</code>	<code> </code>	<code>y</code>
				<code> </code>	<code>z</code>

*This presentation is technically ambiguous (why?)*

We assume application is **left-associative** and has higher precedence than the anonymous function syntax

# Syntax (Unambiguous)

<code>&lt;expr&gt;</code>	<code>::=</code>	<code>fun</code>	<code>&lt;var&gt;</code>	<code>-&gt;</code>	<code>&lt;expr&gt;</code>
					<code>&lt;expr2&gt; { &lt;expr2&gt; }</code>
<code>&lt;expr2&gt;</code>	<code>::=</code>	<code>&lt;var&gt;</code>		<code>(</code>	<code>&lt;expr&gt;</code> <code>)</code>
<code>&lt;var&gt;</code>	<code>::=</code>	<code>a</code>		<code>b</code>	<code>...</code>   <code>y</code>   <code>z</code>

repeat  
`<expr>` 0 or  
more times

*In this grammar we can only use variables or functions in parentheses in applications*

# Syntax (Mathematical)

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

In mathematical settings, we use more compact syntax

Parentheses, precedence, and variables are often left implicit



# Warning: Get used to $\lambda$

$$\lambda x . e \quad \equiv \quad \text{fun } x \text{ -> } e$$

We will use these syntaxes interchangeably  
starting now

*These are the same thing, get used to it*

# Values

$\langle val \rangle ::= \lambda \langle var \rangle . \langle expr \rangle$

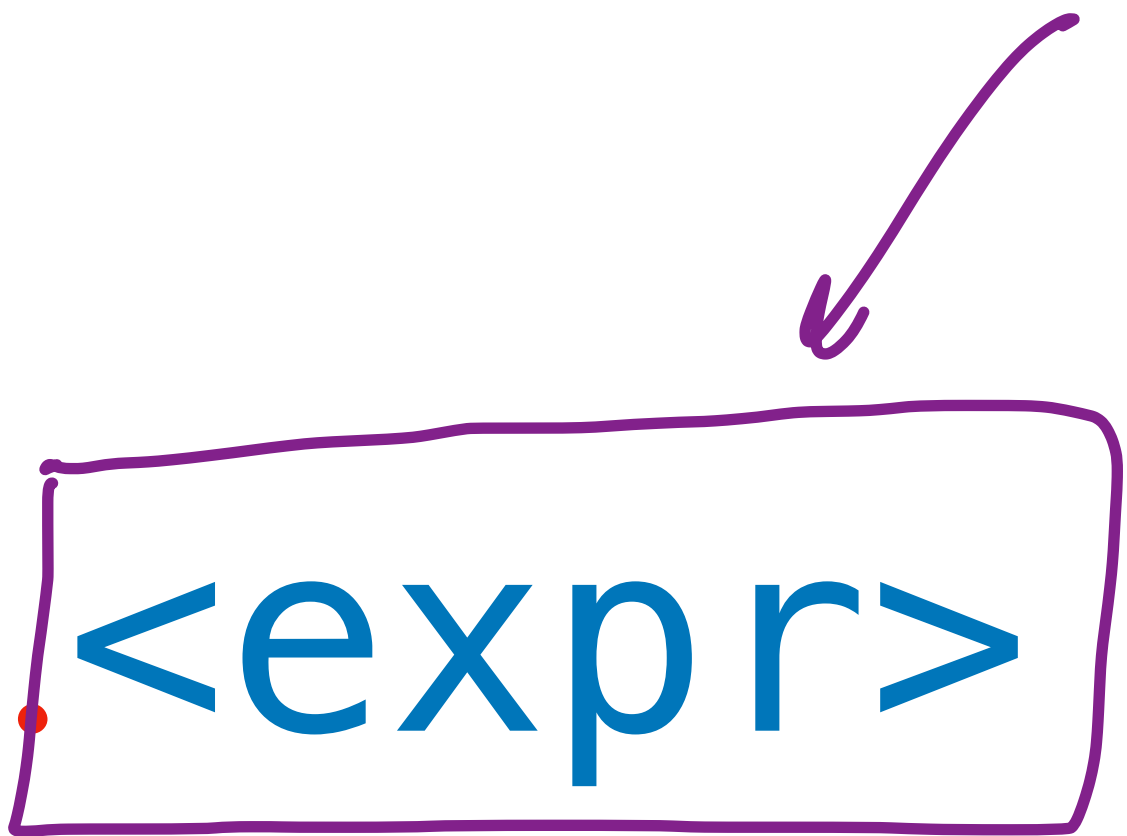
# Values

$\langle val \rangle ::= \lambda \langle var \rangle . \langle expr \rangle$

In arithmetic, values are **numbers**

# Values

$\langle val \rangle ::= \lambda \langle var \rangle \langle expr \rangle$



In arithmetic, values are **numbers**

In the lambda calculus, values are **functions**

# Values

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

In arithmetic, values are **numbers**

In the lambda calculus, values are **functions**

*We often use BNF syntax to specify our values*

# Small-Step Semantics

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$

$$(3) \frac{}{(\lambda x . e)(\lambda y . e') \longrightarrow [(\lambda y . e')/x]e}$$

# Small-Step Semantics

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$

$$(3) \frac{}{(\lambda x . e)(\lambda y . e') \longrightarrow [(\lambda y . e')/x]e}$$

1. We can reduce the LHS of an application

# Small-Step Semantics

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{e_2 \longrightarrow e'_2}{(\lambda x . e_1) e_2 \longrightarrow (\lambda x . e_1) e'_2}$$

$$(3) \frac{}{(\lambda x . e)(\lambda y . e') \longrightarrow [(\lambda y . e')/x]e}$$

1. We can reduce the LHS of an application
2. We can reduce the RHS of an application *if the LHS is already a function*



# Small-Step Semantics

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{e_2 \longrightarrow e'_2}{(\lambda x. e_1) e_2 \longrightarrow (\lambda x. e_1) e'_2}$$

$$(3) \frac{}{(\lambda x. e)(\lambda y. e') \longrightarrow [(\lambda y. e')/x]e}$$

substitution

1. We can reduce the LHS of an application
2. We can reduce the RHS of an application *if the LHS is already a function*
3. We can apply a function to another function by **substitution**. This is also called  **$\beta$ -reduction**

# Example

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2} \quad (2) \frac{e_2 \longrightarrow e'_2}{(\lambda x. e_1) e_2 \longrightarrow (\lambda x. e_1) e'_2}$$

$$(3) \frac{}{(\lambda x. e)(\lambda y. e') \longrightarrow [(\lambda y. e')/x]e}$$

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

$$(\text{fun } f \rightarrow \text{fun } x \rightarrow f \ x) (\text{fun } y \rightarrow y)$$

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

# Small-Step Semantics (Another Form)

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{}{(\lambda x . e) e' \longrightarrow [e' / x] e}$$

# Small-Step Semantics (Another Form)

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{}{(\lambda x . e) e' \longrightarrow [e' / x] e}$$

1. We can reduce the LHS of an application

# Small-Step Semantics (Another Form)

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{}{(\lambda x . e) e' \longrightarrow [e' / x] e}$$

1. We can reduce the LHS of an application
2. We can apply a function to *any* expression via substitution

# Example

$$(1) \frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

$$(2) \frac{}{(\lambda x. e) e' \longrightarrow [e'/x] e}$$

$$(\lambda x. y)((\lambda z. z)(\lambda w. w)) \longrightarrow$$

*never gets evaluated*

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

exercise: evaluate this in first semantics

# Recall: Values

When evaluating, there are **three** "end" cases to evaluation:

- » **value:** we reach the end of our computation and the value of our program
- » **stuck:** we reach an expression that cannot be reduced, but that is not a value
- » **diverge:** the computation never reaches a point where the expression is not reducible

# Stuck Terms

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

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

$$(\lambda x . yx)(\lambda x . x) \longrightarrow$$

$$[\lambda x . x / x] yx = y (\lambda x . x) \not\rightarrow$$

Note: Type systems avoid stuck terms

Based on our operational semantics, it's possible for the above expression to reduce to a value



# Non-Termination

$$\frac{e_1 \longrightarrow e'_1}{e_1 e_2 \longrightarrow e'_1 e_2}$$

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

$$\begin{array}{c} \Omega \\ \parallel \\ (\lambda x. xx)(\lambda x. xx) \longrightarrow \\ \parallel \qquad \parallel \\ \omega \qquad \omega \end{array}$$

$$[\lambda x. xx / x](xx) = (\lambda x. xx)(\lambda x. xx) \rightarrow \Omega \rightarrow \dots$$

And unlike with arithmetic, it's now possible to define expressions which **do not terminate**. These expressions do not have values, but also don't get stuck

# Big-Step Semantics

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

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

# Big-Step Semantics

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

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

1. A function evaluates to a function value (itself)

# Big-Step Semantics

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

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

1. A function evaluates to a function value (itself)
2. If  $e_1$  evaluates to the function  $\lambda x . e$  and  $e_2$  evaluates to the value  $v_2$  and  $e$  with  $v_2$  substituted for  $x$  evaluates to  $v$ , then the application  $e_1 e_2$  evaluates to  $v$

# Big-Step Semantics (Another Form)

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

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

These are the same rules as before except we're not required to evaluate  $e_2$  first

# Big-Step Semantics (Another Form)

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

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

These are the same rules as before except we're not required to evaluate  $e_2$  first

# 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

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

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

# Answer

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



# Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x . e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBV)}$$

$$\frac{e_1 \Downarrow \lambda x . e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBN)}$$

# Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBV)}$$

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBN)}$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

# Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBV)}$$

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBN)}$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

CBV: evaluate the argument of a function *before* substituting it in the function

# Call-by-Value vs. Call-by-Name

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBV)}$$

$$\frac{e_1 \Downarrow \lambda x. e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v} \text{ (CBN)}$$

The two versions of semantics we've given correspond to **call-by-value** (CBV) and **call-by-name** (CBN). These are **evaluation strategies** for functional languages

CBV: evaluate the argument of a function *before* substituting it in the function

CBN: substitute the expression *directly* into the function

# Benefits of CBV

$$\frac{}{\lambda x . e \Downarrow \lambda x . e} \quad \frac{e_1 \Downarrow \lambda x . e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

# Benefits of CBV

$$\frac{}{\lambda x . e \Downarrow \lambda x . e} \quad \frac{e_1 \Downarrow \lambda x . e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

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

# Benefits of CBV

$$\frac{}{\lambda x . e \Downarrow \lambda x . e} \quad \frac{e_1 \Downarrow \lambda x . e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

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

# Benefits of CBV

$$\frac{}{\lambda x . e \Downarrow \lambda x . e} \quad \frac{e_1 \Downarrow \lambda x . e'_1 \quad e_2 \Downarrow v_2 \quad [v_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

$$(\lambda x . x + x + x + x)e \xrightarrow{\text{CBV}} e + e + e + e$$

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)



# Benefits of CBN

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

$$\frac{e_1 \Downarrow \lambda x . e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

# Benefits of CBN

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

$$\frac{e_1 \Downarrow \lambda x . e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

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

# Benefits of CBN

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

$$\frac{e_1 \Downarrow \lambda x . e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

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

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)

# Benefits of CBN

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

$$\frac{e_1 \Downarrow \lambda x . e'_1 \quad [e_2/x]e'_1 \Downarrow v}{e_1 e_2 \Downarrow v}$$

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

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

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)

*Aside.* It's possible to simulate CBN in CBV. Think about it for a bit, ask me after if you're interested

# Side Effects

```
let f x = x + x in  
let y =  
    let _ = print_int 2 in  
    2  
in f y
```

# Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

*What does this program print? **It depends on if we're using CBN or CBV evaluation***

# Side Effects

```
let f x = x + x in  
let y =  
  let _ = print_int 2 in  
  2  
in f y
```

CBV:  
2

CBN:  
2  
2

*What does this program print? **It depends on if we're using CBN or CBV evaluation***

**Definition.** (*informal*) A **side effect** refers to something that happens during the evaluation of a program that is not a part of the formal semantics, e.g., printing to the console

# Side Effects

```
let f x = x + x in
let y =
  let _ = print_int 2 in
  2
in f y
```

*What does this program print? **It depends on if we're using CBN or CBV evaluation***

**Definition.** (*informal*) A **side effect** refers to something that happens during the evaluation of a program that is not a part of the formal semantics, e.g., printing to the console

**Different evaluation strategies yield different side-effectful behavior!**



# **Aside: Evaluation Strategies**

# Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

# Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

Haskell uses **lazy** evaluation also called **call-by-need**

# Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

Haskell uses **lazy** evaluation also called **call-by-need**

In languages with pointers we also often have the option to use **call-by-reference** evaluation or **call-by-sharing**

# Aside: Evaluation Strategies

There are *a lot* more evaluation strategies, all of which optimize *something*

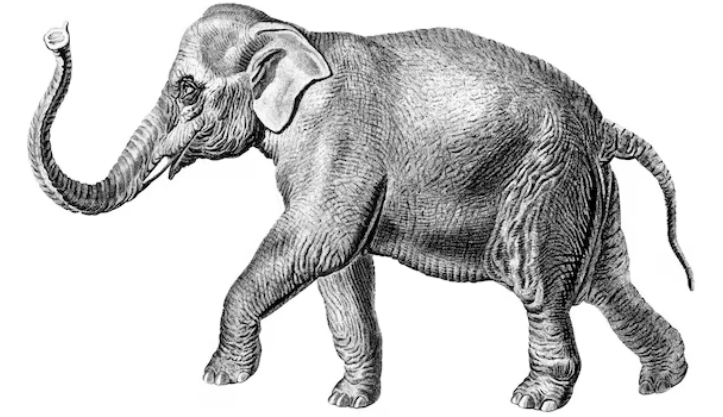
Haskell uses **lazy** evaluation also called **call-by-need**

In languages with pointers we also often have the option to use **call-by-reference** evaluation or **call-by-sharing**

*We will exclusively implement **call-by-value** (because, again, this is what OCaml does)*

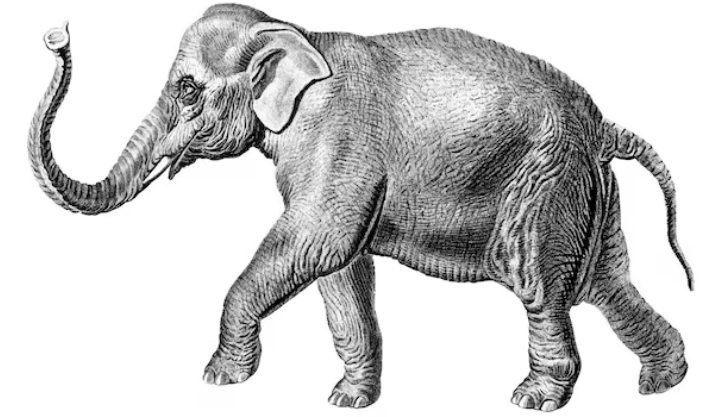
# Substitution

# 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

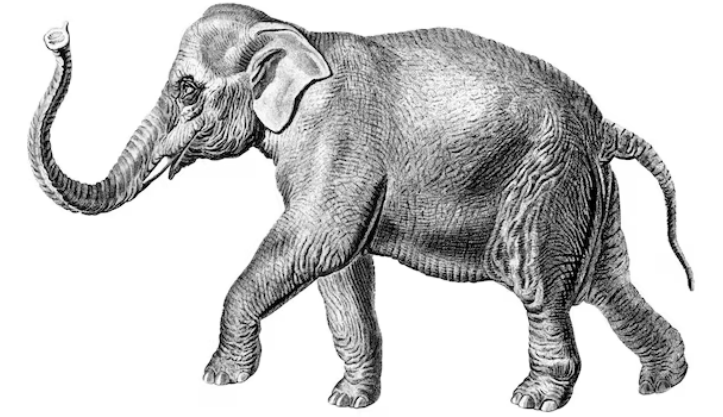


$$\frac{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

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



# Substitution

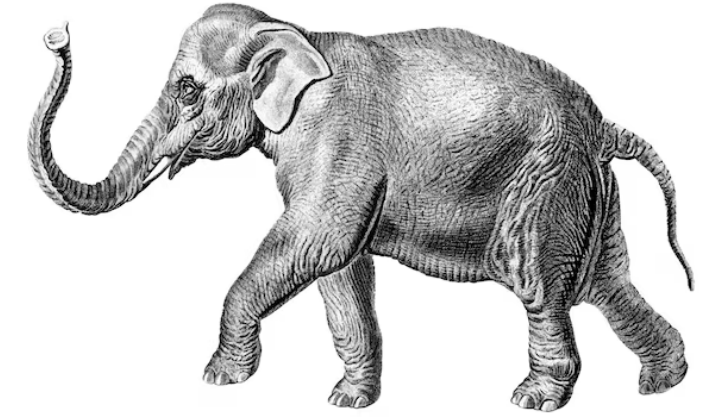


$$\frac{e_1 \Downarrow \lambda x. e \quad e_2 \Downarrow v_2 \quad [v_2/x]e \Downarrow v}{e_1 e_2 \Downarrow v}$$

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...*

# Notation

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

# Notation

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

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

# 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  $v$*

# Notation

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

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...

# The Idea

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

# The Idea

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

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



# The Idea

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

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

OCaml

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

$\lambda$ -calculus

# $\alpha$ -Equivalence

let x = 2 in x + 1

$=_{\alpha}$

let z = 2 in z + 1

OCaml

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

 $=_{\alpha}$ 

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

OCaml

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

$\lambda$ -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  $(\dots \lambda x . e \dots)$

# $\alpha$ -Equivalence

if  $e_1 =_\alpha e_2$  then

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

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

$=_\alpha$

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

OCaml

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

$\lambda$ -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**

# 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)

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

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

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

1. Replace every  $y$  with  $v$ , leave other variables
2. Replace  $y$  with  $v$  in the body of a function
3. Replace  $y$  with  $v$  in both subexpressions of an application

(This is an example of an *inductive definition*)

# Problem Case I

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

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

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

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

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

$$\lambda x. x =_{\alpha} \lambda z. z$$

but

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

$$[y/x](\lambda x. x) \neq_{\alpha} [y/x](\lambda z. z)$$

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)

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

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

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

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

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?**

# Problem Case II

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

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

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

$$\lambda z. x =_a \lambda y. x$$

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

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

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

$$\lambda z. y \neq_a \lambda y. y$$

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

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

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

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

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

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

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

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

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  $e_2$

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

# Definition (Third Attempt)

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

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

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

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

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

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

$$\lambda z . zxz \quad \times$$

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

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

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  $z$ ?**

In mathematics, we can say it's **always possible** to come up with a variable  $z$ , but when we're implementing a programming language, we need an *actual* procedure

# 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 Thursday)

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

*Every closed term is well-scoped*

# One Solution: Well-Scopedness Check

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

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!

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

# demo

(lambda calculus)