

Formal Semantics

Concepts of Programming Languages
Lecture 14

Practice Problem

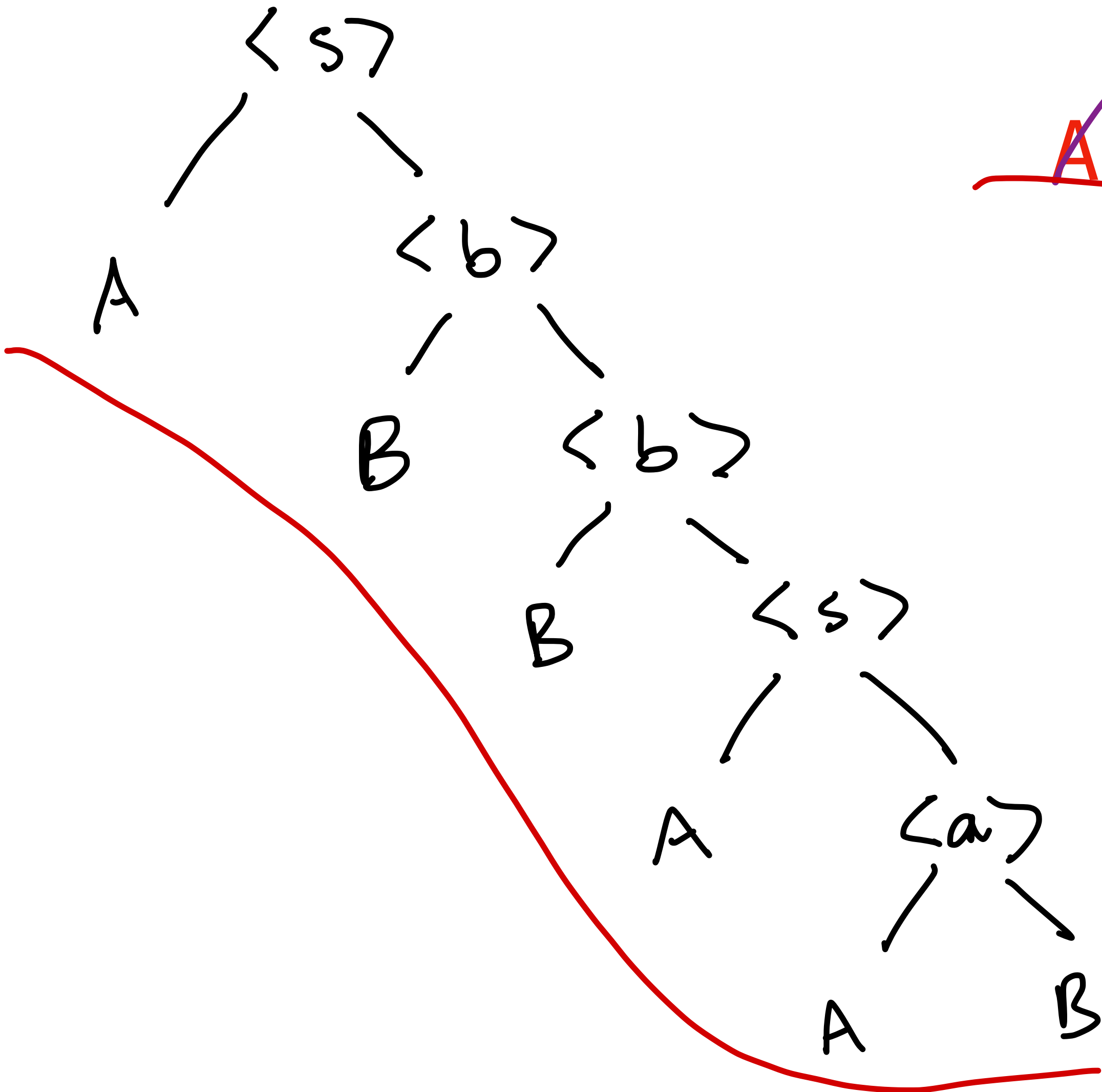
$$\begin{aligned} \langle s \rangle &::= A \langle a \rangle \mid A \langle b \rangle \\ \langle a \rangle &::= A B \\ \langle b \rangle &::= B \langle b \rangle \mid B \langle s \rangle \end{aligned}$$

Is the following sentence recognized by the above grammar?

A B B A A B

Answer

$\langle s \rangle$	$::=$	$A \langle a \rangle \mid A \langle b \rangle$
$\langle a \rangle$	$::=$	$A B$
$\langle b \rangle$	$::=$	$B \langle b \rangle \mid B \langle s \rangle$



~~A~~ ~~B~~ ~~B~~ A A B

Outline

- » Discuss **formal semantics** in general
- » Look at **small-step** and **big-step** semantics with some examples

Introduction

A Thought Experiment

```
x=3
function f () {
    x=2
}
f
echo $x
```

Bash

```
x = 3
def f():
    x = 2
f()
print(x)
```

Python

```
let x = 3
let f () =
    let x = 2 in
    ()
let _ = f ()
let _ = print_int x
```

OCaml

A Thought Experiment

```
x=3
function f () {
    x=2
}
f
echo $x
```

Bash

~~3~~ 2

```
x = 3
def f():
    x = 2
f()
print(x)
```

Python

3

```
let x = 3
let f () =
    let x = 2 in
    ()
let _ = f ()
let _ = print_int x
```

OCaml

3

Question. *How do we know what will happen when a program executes?*

A Thought Experiment

```
x=3
function f () {
    x=2
}
f
echo $x
```

Bash

```
x = 3
def f():
    x = 2
f()
print(x)
```

Python

```
let x = 3
let f () =
    let x = 2 in
    ()
let _ = f ()
let _ = print_int x
```

OCaml

Question. *How do we know what will happen when a program executes?*

Usually we build intuitions by writing programs and reading manuals

A Thought Experiment

```
x=3
function f () {
    x=2
}
f
echo $x
```

Bash

```
x = 3
def f():
    x = 2
f()
print(x)
```

Python

```
let x = 3
let f () =
    let x = 2 in
    ()
let _ = f ()
let _ = print_int x
```

OCaml

Question. *How do we know what will happen when a program executes?*

Usually we build intuitions by writing programs and reading manuals

But many decisions about what it means to execute a program are arbitrary (or based on concerns like efficiency)

Meaning

Meaning

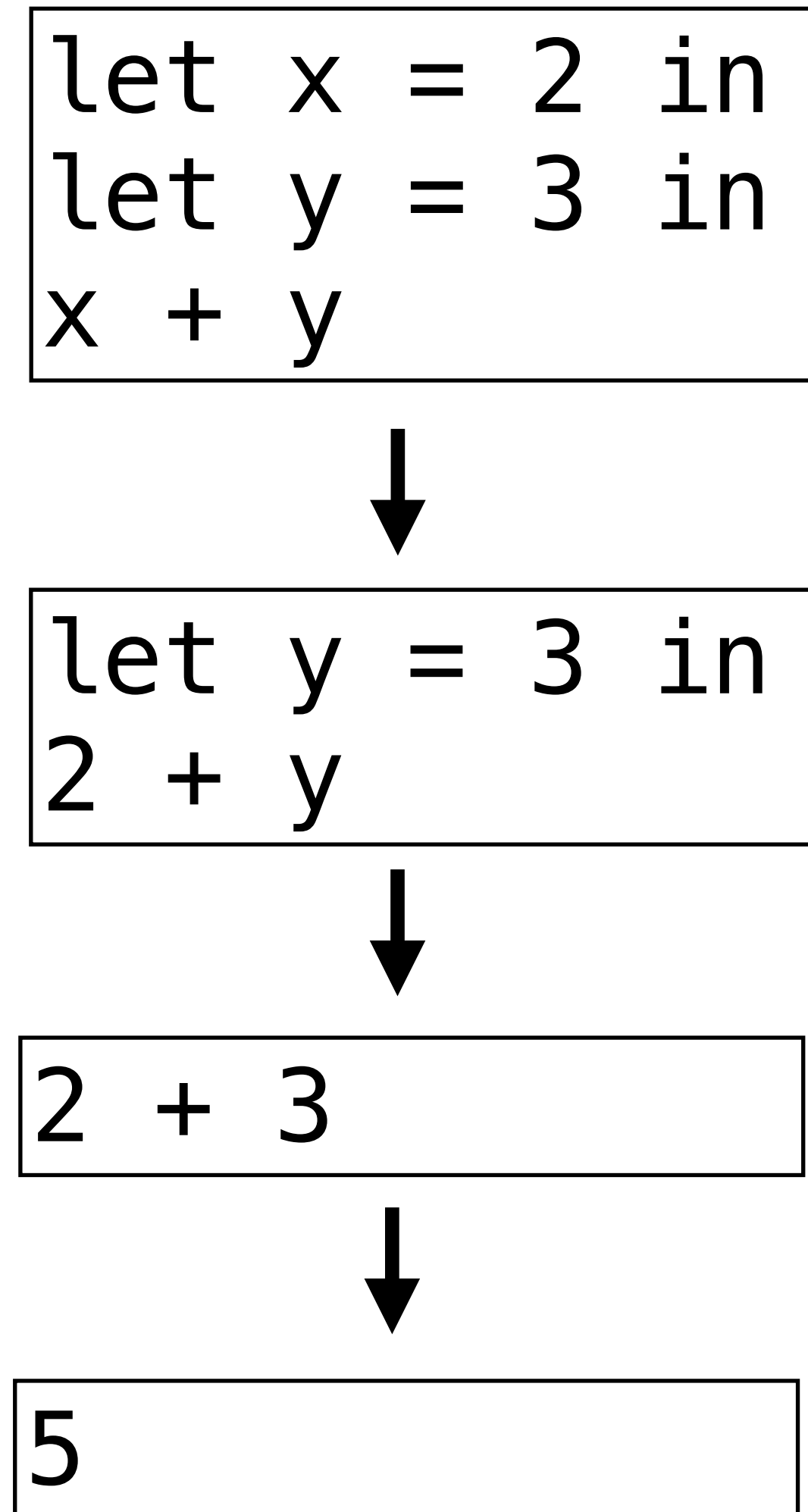
Syntax is interested in the *form*
of a program

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

Meaning

Syntax is interested in the *form* of a program

Semantics is interested in the *meaning* of a program

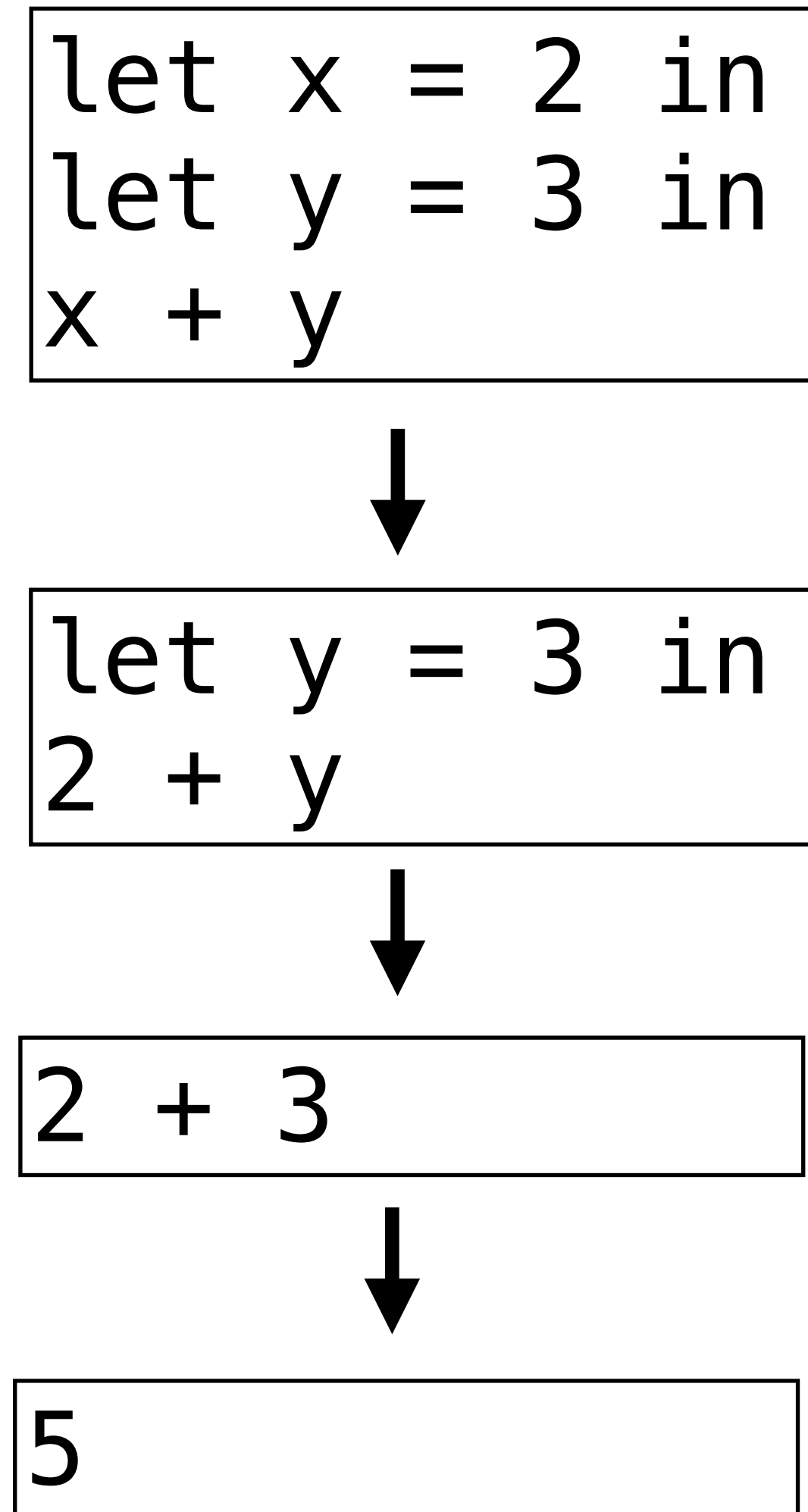


Meaning

Syntax is interested in the *form* of a program

Semantics is interested in the *meaning* of a program

What is the meaning of meaning?



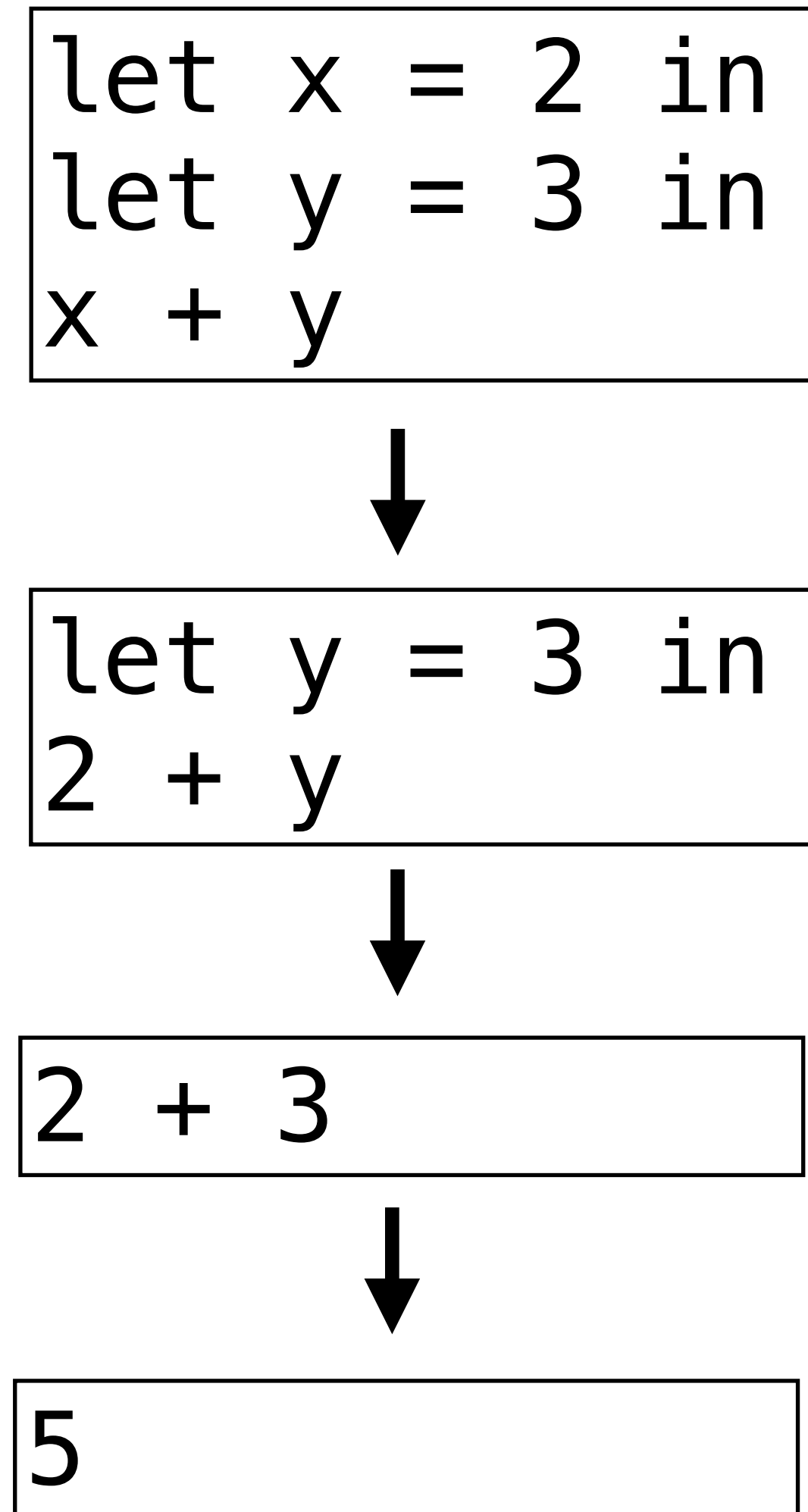
Meaning

Syntax is interested in the *form* of a program

Semantics is interested in the *meaning* of a program

What is the meaning of meaning?

Formal semantics is the mathematical study of meaning



Aside: Denotational vs. Operational Semantics

Aside: Denotational vs. Operational Semantics

Denotational semantics is interested in what a syntactic object "denotes" i.e. in interpreting programs as *objects in a mathematical space*

$$1 + 2 * 3 + 4 = 11$$

$$1 + 12 - 2 = 11$$

Aside: Denotational vs. Operational Semantics

Denotational semantics is interested in what a syntactic object "denotes" i.e. in interpreting programs as *objects in a mathematical space*

$$\begin{aligned}1 + 2 * 3 + 4 &= 11 \\1 + 12 - 2 &= 11\end{aligned}$$

Operational semantics is interested in how a programming language "operates" i.e. how a program *behaves* during execution

$$\begin{aligned}1 + 2 * 3 + 4 &\longrightarrow 1 + 6 + 4 \\&\longrightarrow 7 + 4 \\&\longrightarrow 11\end{aligned}$$

Aside: Denotational vs. Operational Semantics

Denotational semantics is interested in what a syntactic object "denotes" i.e. in interpreting programs as *objects in a mathematical space*

$$\begin{aligned}1 + 2 * 3 + 4 &= 11 \\ 1 + 12 - 2 &= 11\end{aligned}$$

Operational semantics is interested in how a programming language "operates" i.e. how a program *behaves* during execution

This course

$$\begin{aligned}1 + 2 * 3 + 4 &\longrightarrow 1 + 6 + 4 \\ &\longrightarrow 7 + 4 \\ &\longrightarrow 11\end{aligned}$$

Small-Step vs. Big-Step Semantics

Small-Step vs. Big-Step Semantics

Small-step operational semantics is interested in *program transformation*, i.e., how a program transforms "one step at a time"

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



```
let y = 3 in  
2 + y
```



```
2 + 3
```



```
5
```

Small-Step vs. Big-Step Semantics

Small-step operational semantics is interested in *program transformation*, i.e., how a program transforms "one step at a time"

Big-step operational semantics is interested in *evaluation*, i.e., what is the value of the program once a program has finished evaluating

$$\frac{2 \Downarrow 2}{\text{let } x = 2 \text{ in } \frac{\frac{3 \Downarrow 3}{\text{let } y = 3 \text{ in } 2 + y \Downarrow 5}}{2 + 3 \Downarrow 5}} \Downarrow 5$$

Small-Step vs. Big-Step Semantics

Small-step operational semantics is interested in *program transformation*, i.e., how a program transforms "one step at a time"

Mini-projects

Big-step operational semantics is interested in *evaluation*, i.e., what is the value of the program once a program has finished evaluating

$$\frac{2 \Downarrow 2}{\text{let } x = 2 \text{ in } \frac{\frac{3 \Downarrow 3}{\text{let } y = 3 \text{ in } 2 + y \Downarrow 5}}{2 + 3 \Downarrow 5}} \Downarrow 5$$

Static vs. Dynamic Semantics

Static vs. Dynamic Semantics

Static semantics

refers to the meaning
given to a program
before it is evaluated

```
% ocaml silly.ml
```

```
File "./silly.ml", line 1, characters 8-9:
```

```
1 | let x = 2 +. 3.
```

^

Error: This expression has type int but an expression was
expected of type
float

Hint: Did you mean `2.'?

Static vs. Dynamic Semantics

Static semantics

refers to the meaning
given to a program
before it is evaluated

Dynamic semantics

refers to the behavior
of a program *during*
evaluation

```
% ocaml silly.ml
```

```
File "./silly.ml", line 1, characters 8-9:
```

```
1 | let x = 2 +. 3.
```

^

Error: This expression has type int but an expression was
expected of type
float

Hint: Did you mean `2.'?

```
utop # let x = 2 + 3;;
```

```
val x : int = 5
```

Static vs. Dynamic Semantics

Type checking

Static semantics

refers to the meaning
given to a program
before it is evaluated

```
% ocaml silly.ml
```

```
File "./silly.ml", line 1, characters 8-9:
```

```
1 | let x = 2 +. 3.
```

^

Error: This expression has type int but an expression was
expected of type
float

Hint: Did you mean `2.'?

Dynamic semantics

refers to the behavior
of a program *during*
evaluation

```
utop # let x = 2 + 3;;
```

```
val x : int = 5
```

Static vs. Dynamic Semantics

Type checking

Static semantics

refers to the meaning
given to a program
before it is evaluated

```
% ocaml silly.ml
```

```
File "./silly.ml", line 1, characters 8-9:
```

```
1 | let x = 2 +. 3.
```

^

Error: This expression has type int but an expression was
expected of type
float

Hint: Did you mean `2.'?

Evaluation

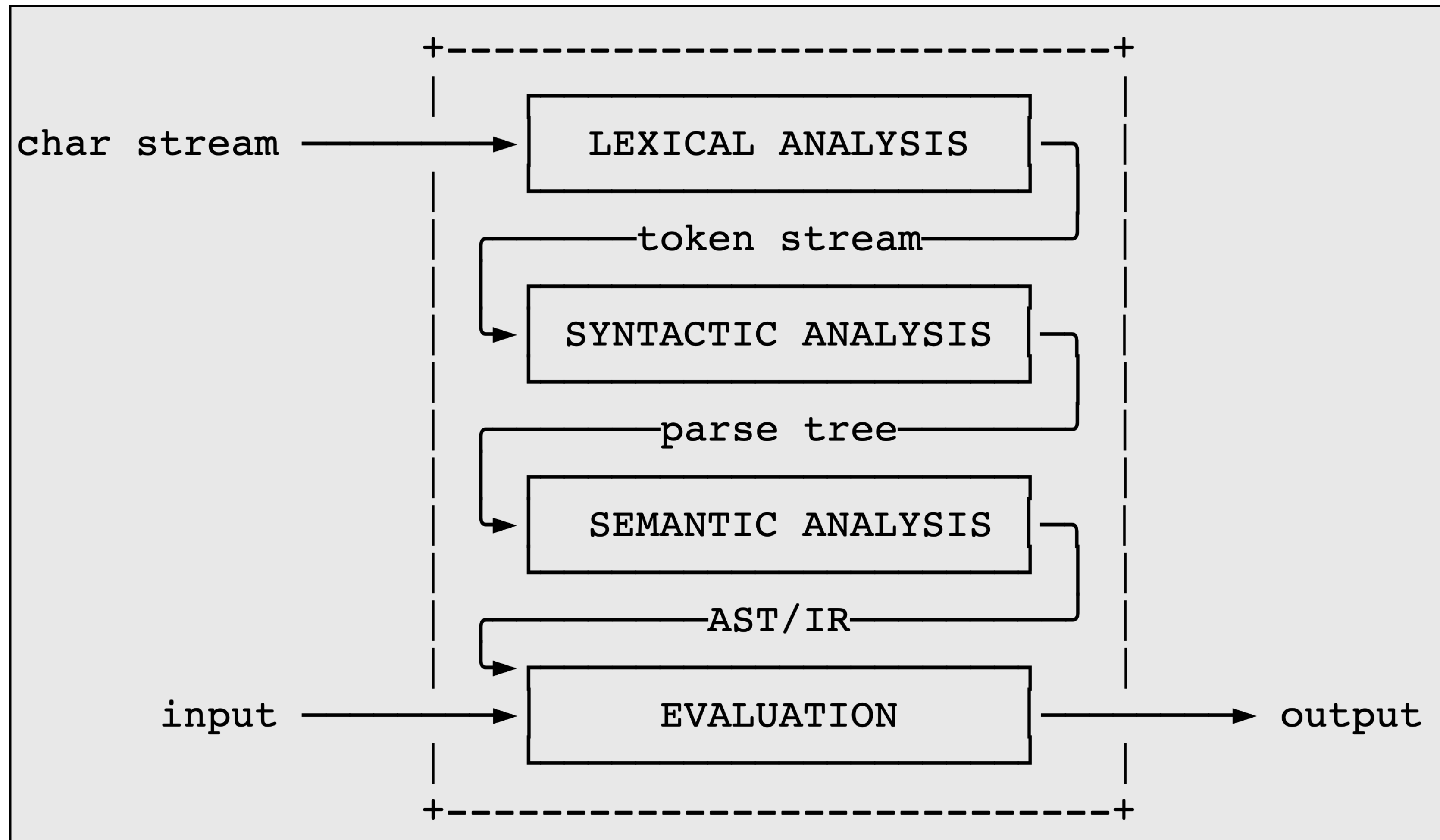
Dynamic semantics

refers to the behavior
of a program *during*
evaluation

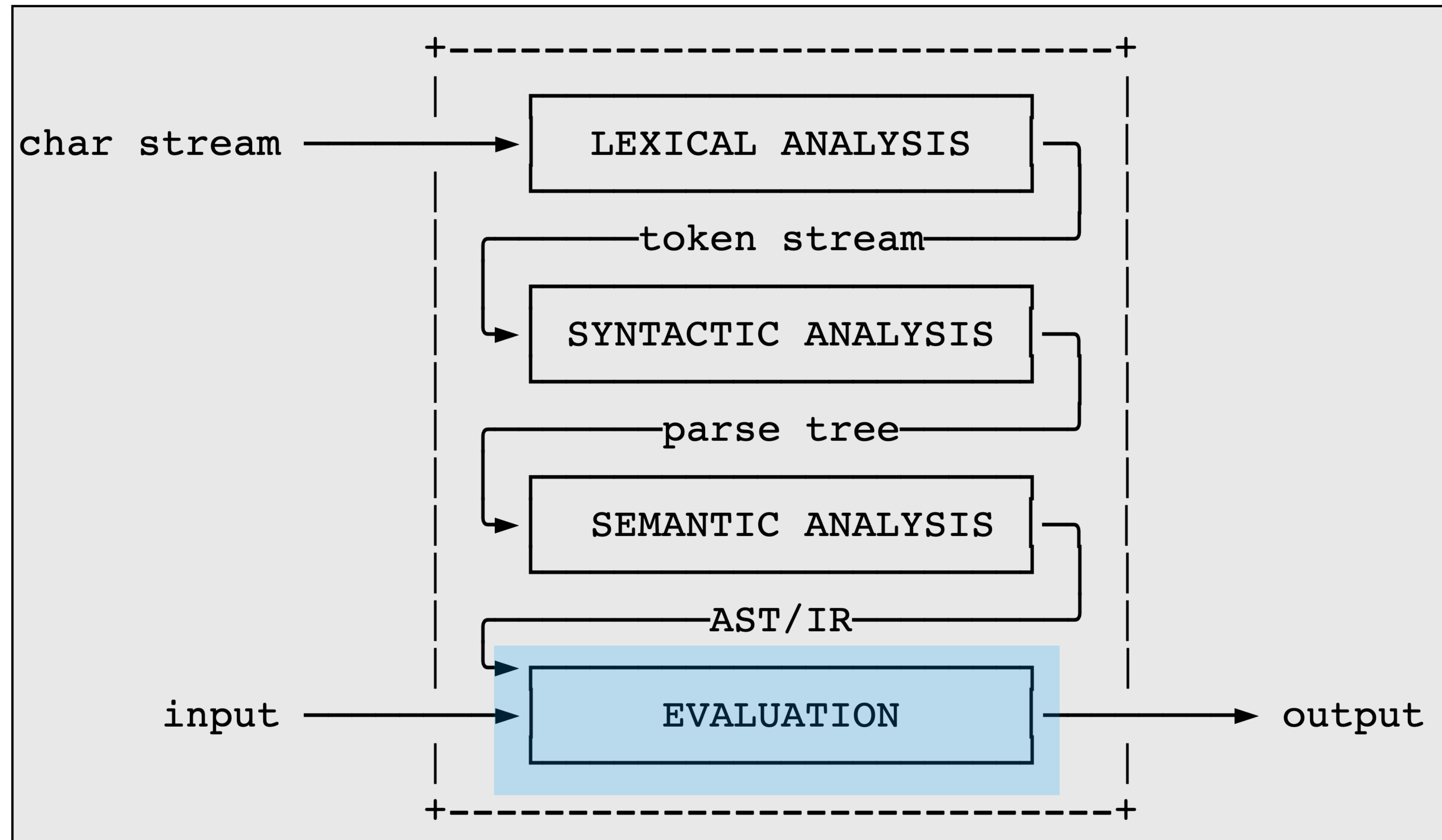
```
utop # let x = 2 + 3;;
```

```
val x : int = 5
```

Recall: The Picture

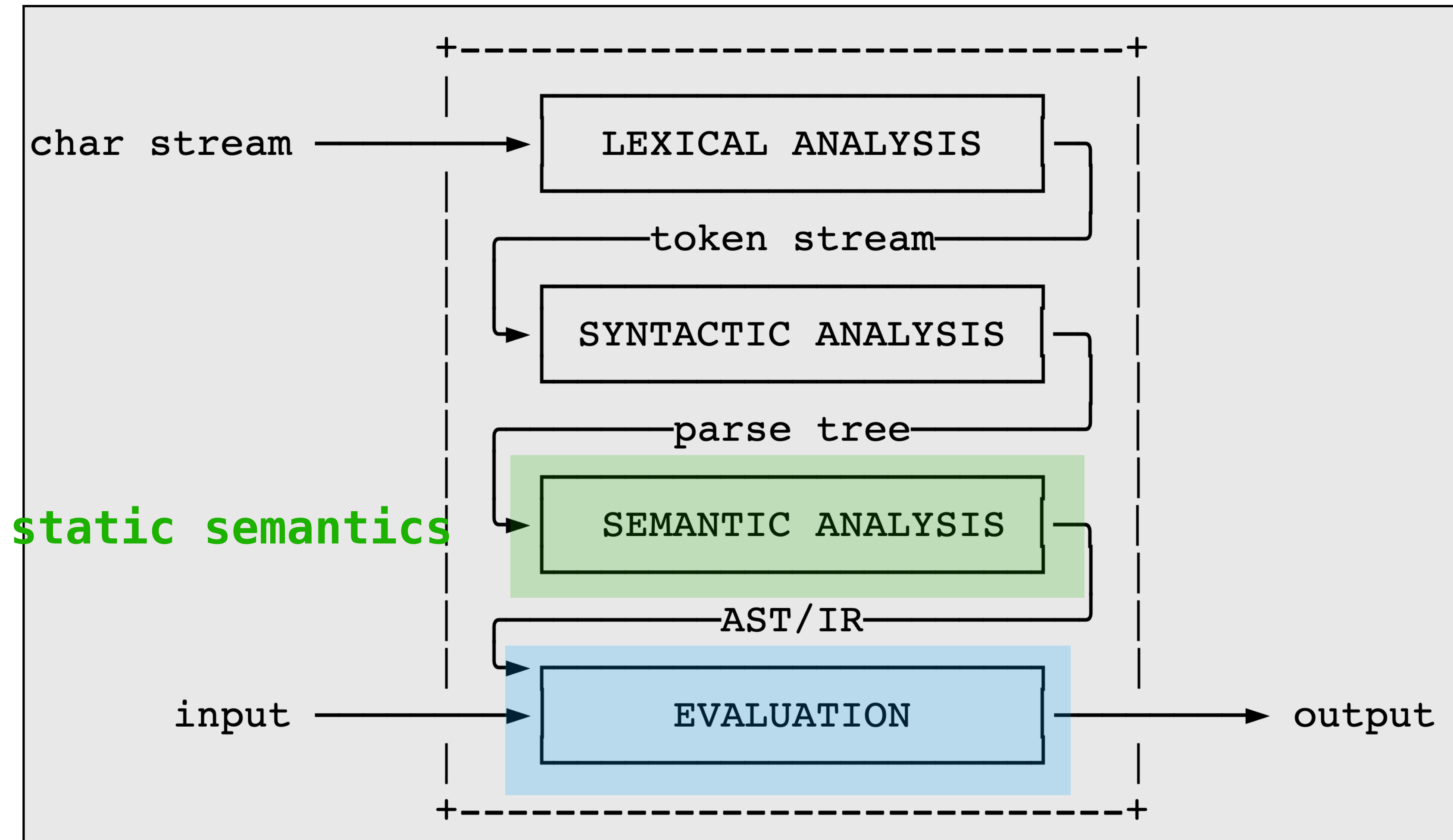


Recall: The Picture



dynamic semantics (this week + next week)

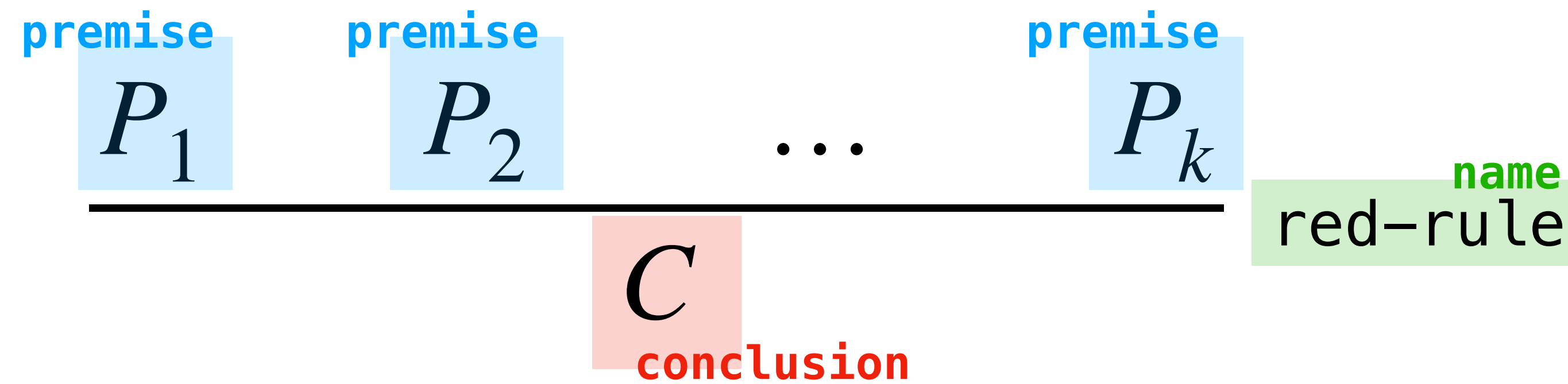
Recall: The Picture



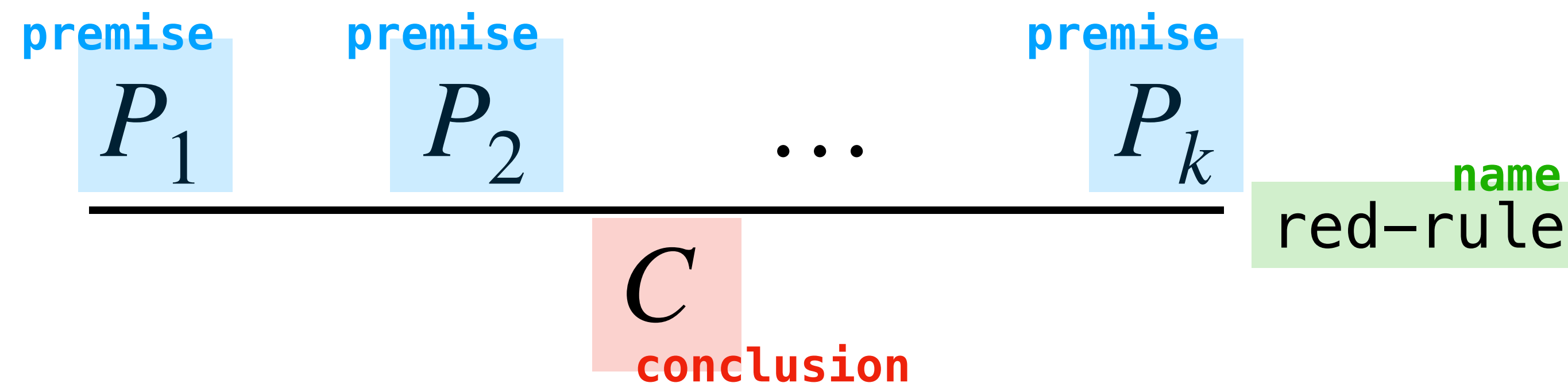
dynamic semantics (this week + next week)

Operational Semantics

Recall: Inference Rules

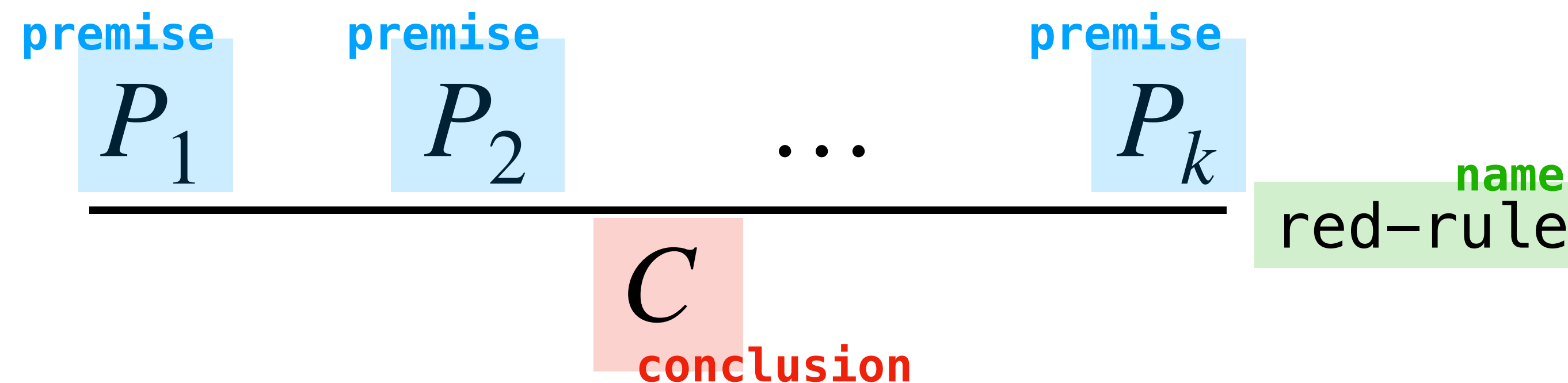


Recall: Inference Rules



Then general form of a reduction rule has a collection of **premises** and a **conclusion**

Recall: Inference Rules



Then general form of a reduction rule has a collection of **premises** and a **conclusion**

There may be no premises, this is called an **axiom**

Example

$$\frac{e_1 \xrightarrow{\text{premise}} e'_1}{(\text{add } e_1 \ e_2) \xrightarrow{\text{conclusion}} (\text{add } e'_1 \ e_2)} \text{add-left}$$

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

Example Programs:

```
(add 2 3)  
(add (add 2 3) 5)  
(eq (add 2 3) (sub 7 2))  
(add true 2)
```

Example

$$\frac{e_1 \xrightarrow{\text{premise}} e'_1}{(\text{add } e_1 \ e_2) \xrightarrow{\text{add-left}} (\text{add } e'_1 \ e_2)} \text{conclusion}$$

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

Example Programs:

```
(add 2 3)  
(add (add 2 3) 5)  
(eq (add 2 3) (sub 7 2))  
(add true 2)
```

If e_1 reduces to e'_1 in one step, then $\text{add } e_1 \ e_2$ reduces to $\text{add } e'_1 \ e_2$ in one step

Another Example

n_1 is a number

n_2 is a number

$(\text{add } n_1 \ n_2) \longrightarrow n_1 + n_2$

add-ok

If n_1 and n_2 are numbers then $(\text{add } n_1 \ n_2)$ reduces in one step to the number $n_1 + n_2$

In this case, the premises are side-conditions

(We'll come back to these examples)

Small-Step Semantics

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

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

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

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

Small-Step Semantics

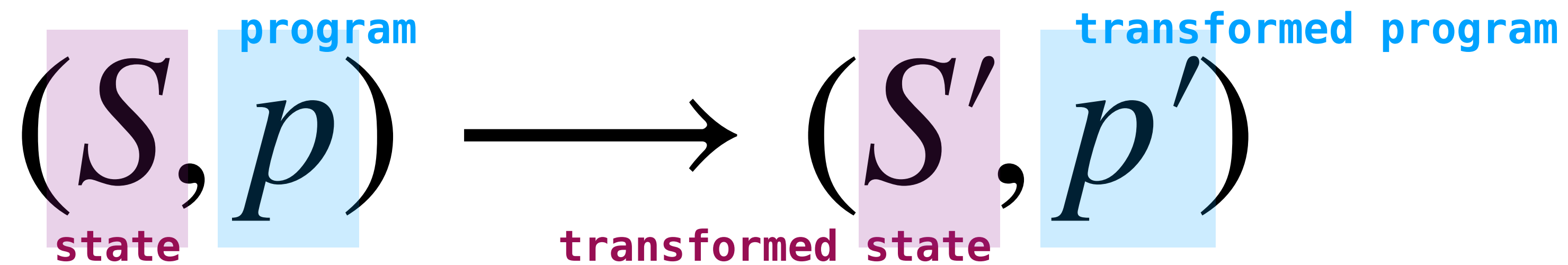
$$(S, \overset{\text{program}}{p}) \longrightarrow (S', \overset{\text{transformed program}}{p'})$$

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

Small-Step Semantics

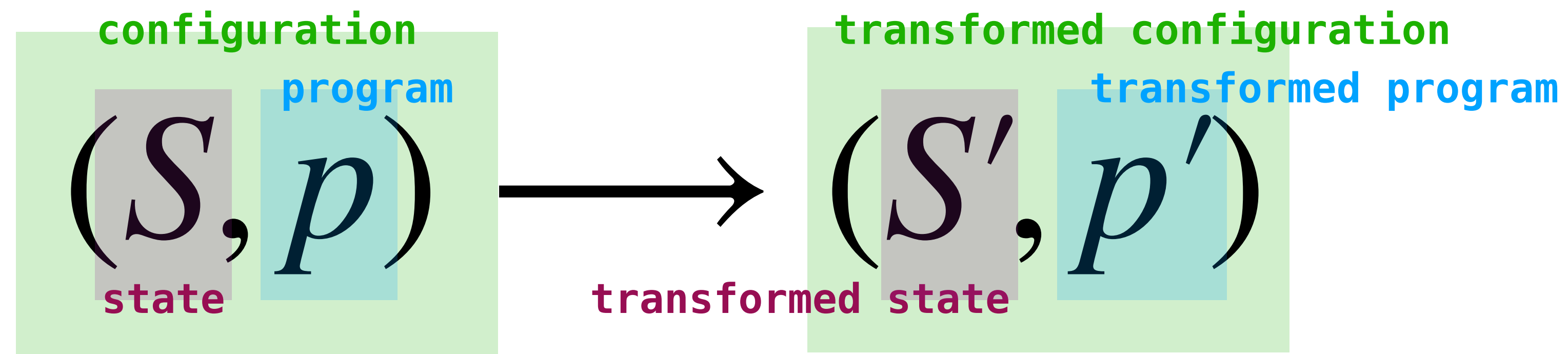


Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

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

Small-Step Semantics

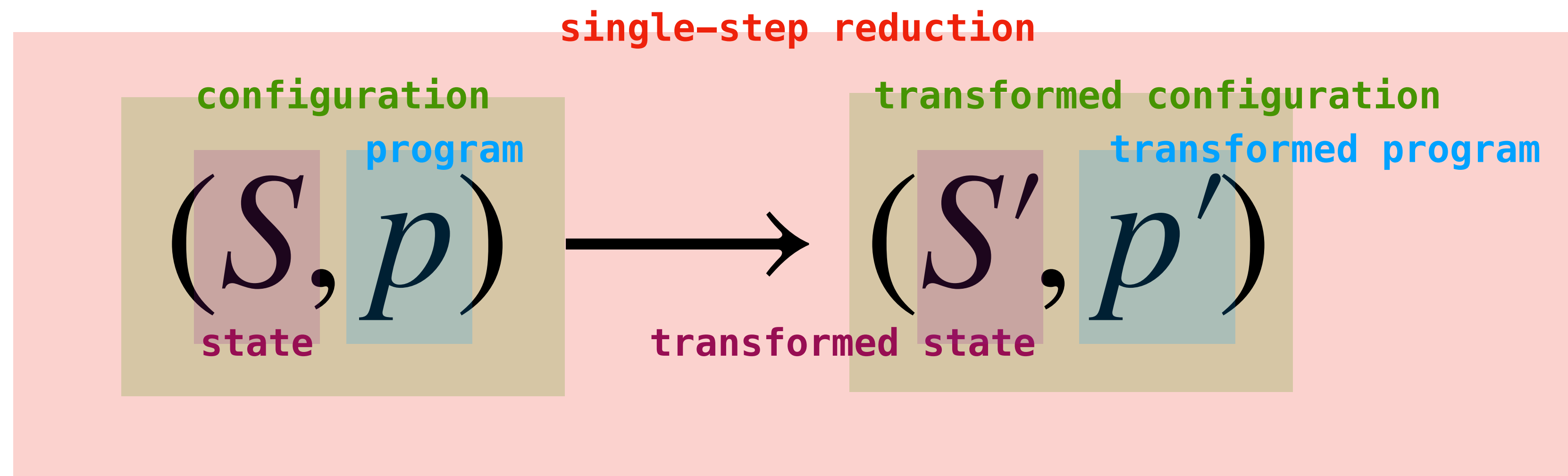


Small-step semantics formalizes a "**step by step**" computation which **reduces** a syntactic object until no reductions can be done

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

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

Example: Arithmetic Expressions

$$\left(\underbrace{\emptyset}_{\text{state}}, \underbrace{10 \times (2 + 3)}_{\text{program}} \right) \longrightarrow (\emptyset, 10 \times 5) \longrightarrow (\emptyset, 50)$$

State: none

Program: arithmetic expression

Example: (Fragment of) OCaml

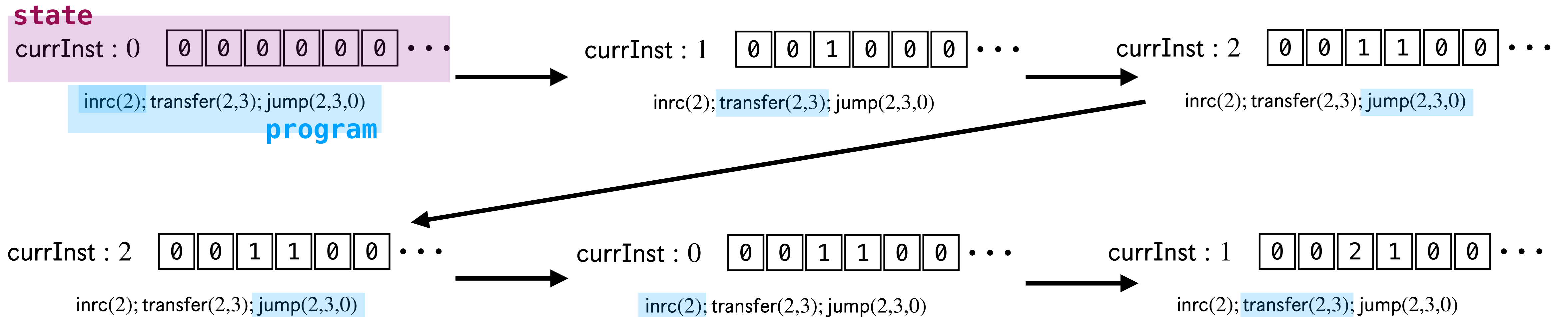
$(\emptyset, \text{let } x = 3 \text{ in if } x > 10 \text{ then } 4 \text{ else } 5)$ $\longrightarrow (\emptyset, \text{if } 3 > 10 \text{ then } 4 \text{ else } 5)$
 $\longrightarrow (\emptyset, \text{if false then } 4 \text{ else } 5)$
 $\longrightarrow (\emptyset, 5)$

State: none

Program: OCaml expression

For purely functional languages
there is no state

Example: Unlimited Register Machines



State: (current instruction pointer) +
 (collection of number registers)

Program: sequence of commands for updating registers
values and current instruction

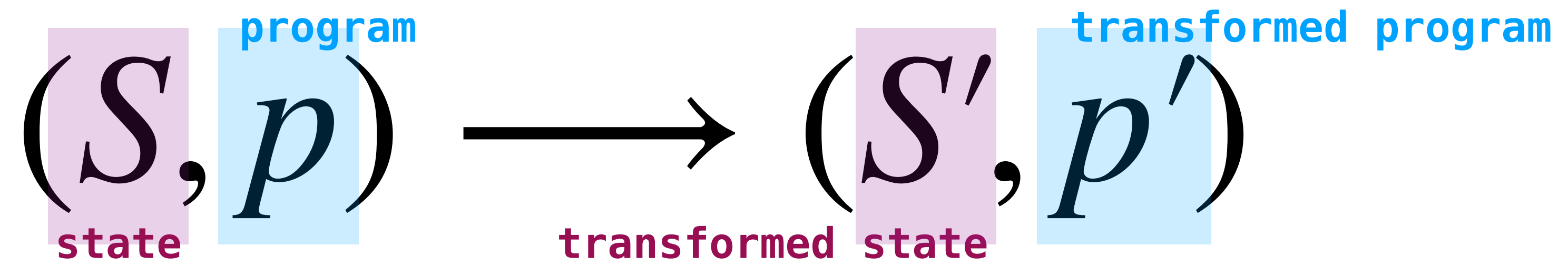
Example: Stack-Oriented Language

^{state}
(\emptyset , ^{program} push 2; push 3; add) \longrightarrow
(2 :: \emptyset , push 3; add) \longrightarrow
(3 :: 2 :: \emptyset , add) \longrightarrow
(5 :: \emptyset , ϵ)

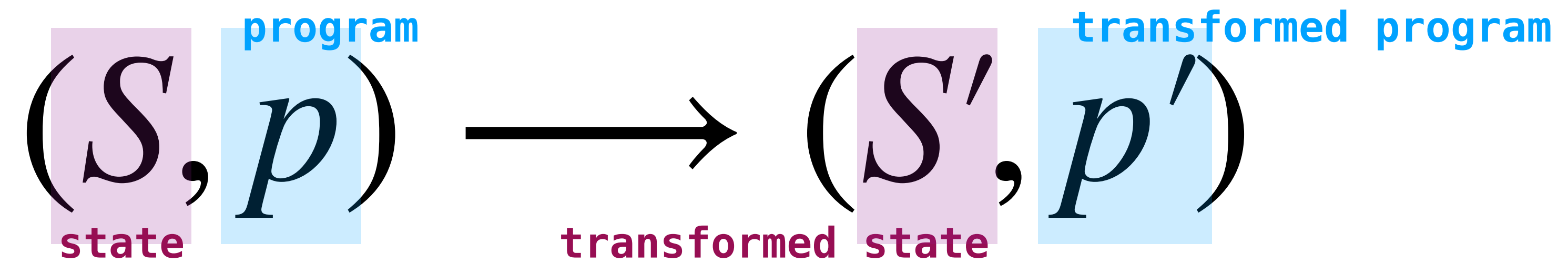
State: stack (i.e., list) of values

Program: sequence of commands for manipulating the stack

High-Level

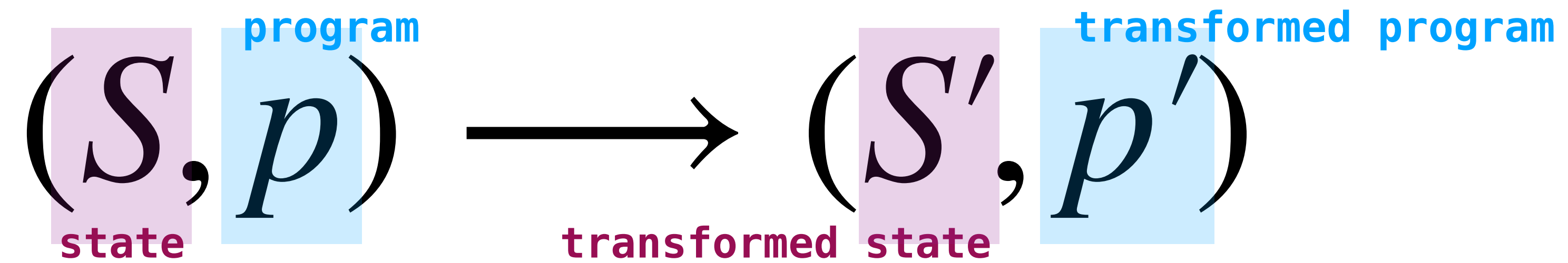


High-Level



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

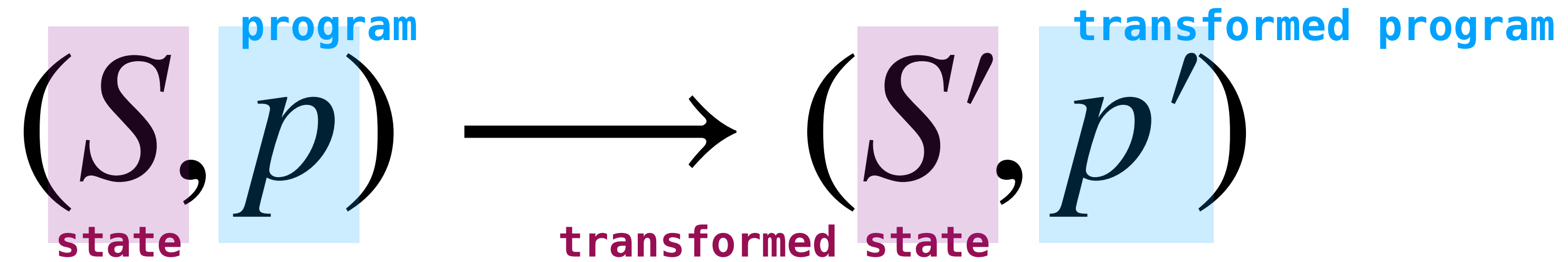
High-Level



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

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

High-Level



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

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

Example

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

Example

```

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

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left}$$

$$\frac{e_2 \longrightarrow e'_2}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e_1 \ e'_2)} \text{ add-right}$$

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

$(+ (+ 2 3) (+ 4 5)) \longrightarrow$

$(+ 5 (+ 4 5)) \longrightarrow$

$(+ 5 9) \longrightarrow 14$

Example

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

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left}$$

$$\frac{e_2 \longrightarrow e'_2}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e_1 \ e'_2)} \text{ add-right}$$

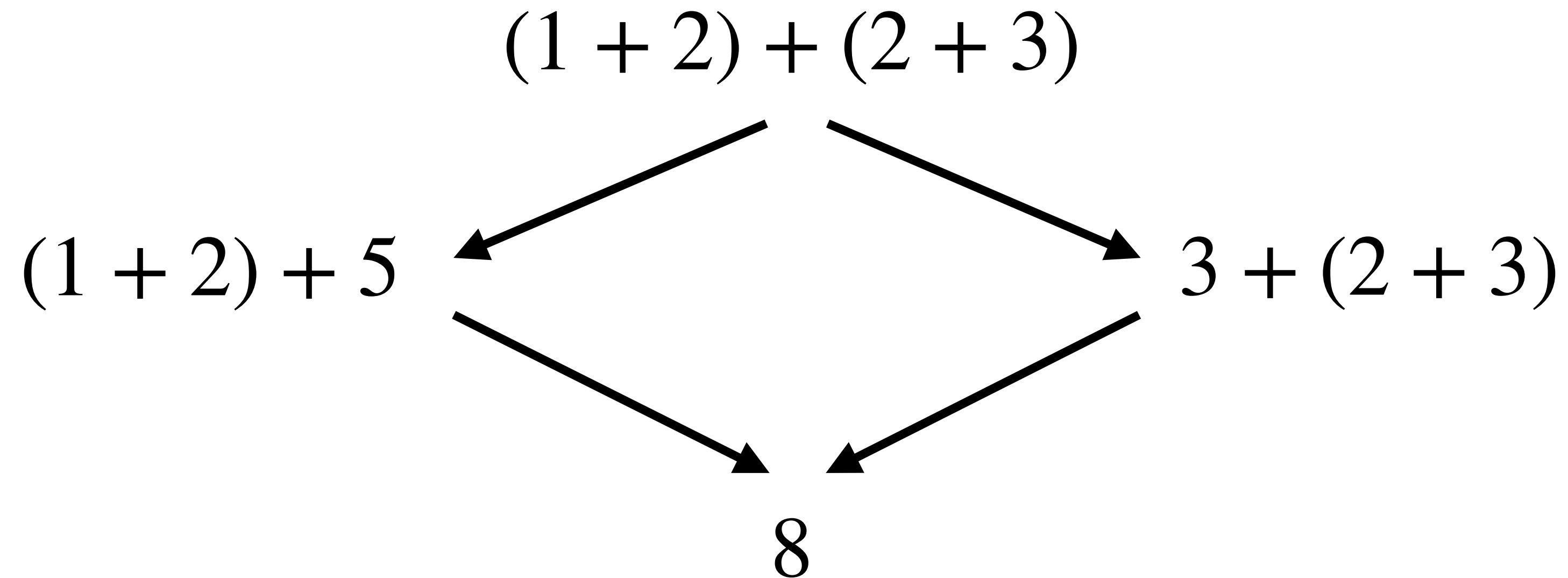
$$\frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{add } n_1 \ n_2) \longrightarrow n_1 + n_2} \text{ add-ok}$$

$$\frac{e_1 \longrightarrow e'_1}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e'_1 \ e_2)} \text{ sub-left}$$

$$\frac{e_2 \longrightarrow e'_2}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e_1 \ e'_2)} \text{ sub-right}$$

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

Reduction is a Relation



It's important to recognize that **reduction is a *relation***

This means there may be **multiple choices of reductions**

When possible, we try to design our rules to avoid this

Reduction is a Relation

$$\frac{\text{add } 1 \ 2 \longrightarrow 3}{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } 3 \ (\text{add } 2 \ 3))} \text{ add-left}$$

$$\frac{\text{add } 2 \ 3 \longrightarrow 5}{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } (\text{add } 1 \ 2) \ 5)} \text{ add-right}$$

Reduction is a Relation

$$\frac{\text{add } 1 \ 2 \longrightarrow 3}{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } 3 \ (\text{add } 2 \ 3))} \text{ add-left}$$

$$\frac{\text{add } 2 \ 3 \longrightarrow 5}{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } (\text{add } 1 \ 2) \ 5)} \text{ add-right}$$

There are two reductions from `(add (add 1 2) (add 2 3))` in our current rule set

Reduction is a Relation

$$\frac{\text{add } 1 \ 2 \longrightarrow 3}{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } 3 \ (\text{add } 2 \ 3))} \text{ add-left}$$

$$\frac{\text{add } 2 \ 3 \longrightarrow 5}{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } (\text{add } 1 \ 2) \ 5)} \text{ add-right}$$

There are two reductions from `(add (add 1 2) (add 2 3))` in our current rule set

We can avoid this by *breaking symmetry*. We will enforce that the right argument can be reduced only when the `left argument is completely reduced`

Example: Addition

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

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left}$$

$$\frac{v \text{ is a number} \quad e_2 \longrightarrow e'_2}{(\text{add } v \ e_2) \longrightarrow (\text{add } v \ e'_2)} \text{ add-right}$$

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

Enforcing an Evaluation Order

$$\frac{\text{add } 1 \ 2 \longrightarrow 3}{(\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3)) \longrightarrow (\text{add } 3 (\text{add } 2 \ 3))} \text{ add-left}$$

$$\frac{\text{add } 2 \ 3 \longrightarrow 5}{(\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3)) \longrightarrow (\text{add } (\text{add } 1 \ 2) 5)} \text{ add-right}$$

↑
not a number

The new rule enforces that arguments of **add** are evaluated from left to right

Example

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

Example

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

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left}$$

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

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

Example

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

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left}$$

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

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

$$\frac{e_1 \longrightarrow e'_1}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e'_1 \ e_2)} \text{ sub-left}$$

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

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

Practice Problem

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

*Write down the reduction rules for **eq** (to the best of your ability) so that the left argument is evaluated before the right argument*

Answer

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

Two Questions

Two Questions

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

Two Questions

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

» Show that $C \longrightarrow C'$

Two Questions

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

- » Show that $C \longrightarrow C'$
- » Given C , determine a configuration C' such that $C \longrightarrow C'$ (and show that it holds)

Two Questions

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

» Show that $C \longrightarrow C'$

» Given C , determine a configuration C' such that $C \longrightarrow C'$ (and show that it holds)

Recall: Derivations

$$\frac{\frac{\frac{}{(add\ 1\ 2) \longrightarrow 3} \text{ add-ok}}{(add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow (add\ 3\ (add\ 2\ 3))} \text{ add-left}}{sub\ 10\ (add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow sub\ 10\ (add\ 3\ (add\ 2\ 3))} \text{ sub-right}$$

Recall: Derivations

$$\frac{\frac{\frac{}{(add\ 1\ 2) \longrightarrow 3} \text{ add-ok}}{(add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow (add\ 3\ (add\ 2\ 3))} \text{ add-left}}{sub\ 10\ (add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow sub\ 10\ (add\ 3\ (add\ 2\ 3))} \text{ sub-right}$$

A **derivation** is a tree of reductions, gotten by applying reduction rules. The leaves are trivial premises

Recall: Derivations

$$\frac{\frac{\frac{}{(add\ 1\ 2) \longrightarrow 3} \text{ add-ok}}{(add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow (add\ 3\ (add\ 2\ 3))} \text{ add-left}}{sub\ 10\ (add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow sub\ 10\ (add\ 3\ (add\ 2\ 3))} \text{ sub-right}$$

A **derivation** is a tree of reductions, gotten by applying reduction rules. The leaves are trivial premises

A derivation is a **proof** that the reduction step is valid in the operational semantics

Recall: Derivations

$$\frac{\frac{\frac{}{(add\ 1\ 2) \longrightarrow 3} \text{ add-ok}}{(add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow (add\ 3\ (add\ 2\ 3))} \text{ add-left}}{sub\ 10\ (add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow sub\ 10\ (add\ 3\ (add\ 2\ 3))} \text{ sub-right}$$

A **derivation** is a tree of reductions, gotten by applying reduction rules. The leaves are trivial premises

A derivation is a **proof** that the reduction step is valid in the operational semantics

We've done this!

Recall: Building Derivations

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))

Recall: Building Derivations

`sub 10 (add (add 1 2) (add 2 3)) → sub 10 (add 3 (add 2 3))`

We can build derivations from the ground up, applying rules in reverse

Recall: Building Derivations

`sub 10 (add (add 1 2) (add 2 3)) → sub 10 (add 3 (add 2 3))`

We can build derivations from the ground up, applying rules in reverse

If the reduction is valid, then at each step we should be able to find a rule to apply

Recall: Building Derivations

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))

We can build derivations from the ground up, applying rules in reverse

If the reduction is valid, then at each step we should be able to find a rule to apply

Recall: Building Derivations

$$\frac{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } 3 \ (\text{add } 2 \ 3))}{\text{sub } 10 \ (\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow \text{sub } 10 \ (\text{add } 3 \ (\text{add } 2 \ 3))} \text{sub-right}$$

We can build derivations from the ground up, applying rules in reverse

If the reduction is valid, then at each step we should be able to find a rule to apply

Recall: Building Derivations

$$\frac{(\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow (\text{add } 3 \ (\text{add } 2 \ 3))}{\text{sub } 10 \ (\text{add } (\text{add } 1 \ 2) \ (\text{add } 2 \ 3)) \longrightarrow \text{sub } 10 \ (\text{add } 3 \ (\text{add } 2 \ 3))} \text{sub-right}$$

We can build derivations from the ground up, applying rules in reverse

If the reduction is valid, then at each step we should be able to find a rule to apply

Recall: Building Derivations

$$\frac{\frac{\text{sub } 10 \text{ (add (add 1 2) (add 2 3))} \longrightarrow \text{sub } 10 \text{ (add 3 (add 2 3))}}{\text{(add (add 1 2) (add 2 3))} \longrightarrow \text{(add 3 (add 2 3))}} \text{ sub-right}$$

add-left

(Note: In the original image, the sub-expression (add 1 2) is highlighted in light blue in the left-hand side of the middle step, and the result 3 is highlighted in light blue in the right-hand side. Purple boxes and arrows indicate the substitution of (add 1 2) with 3.)

We can build derivations from the ground up, applying rules in reverse

If the reduction is valid, then at each step we should be able to find a rule to apply

Recall: Building Derivations

$$\frac{\frac{\frac{}{(add\ 1\ 2) \longrightarrow 3} \text{ add-ok}}{(add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow (add\ 3\ (add\ 2\ 3))} \text{ add-left}}{sub\ 10\ (add\ (add\ 1\ 2)\ (add\ 2\ 3)) \longrightarrow sub\ 10\ (add\ 3\ (add\ 2\ 3))} \text{ sub-right}$$

We can build derivations from the ground up, applying rules in reverse

If the reduction is valid, then at each step we should be able to find a rule to apply

Two Questions

Once we have a small-step semantics, there are **two questions** we can ask (as PL designers and on the final exam):

» Show that $C \longrightarrow C'$

» Given C , determine a configuration C' such that $C \longrightarrow C'$ (and show that it holds)

Single-Step Evaluation

$(\text{sub } 10 (\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3))) \longrightarrow ???$

Single-Step Evaluation

`(sub 10 (add (add 1 2) (add 2 3))) → ???`

The more "realistic" situation is to be given a program and then try to `figure out what it evaluates to` in a single step

Single-Step Evaluation

`(sub 10 (add (add 1 2) (add 2 3))) → ???`

The more "realistic" situation is to be given a program and then try to `figure out what it evaluates to` in a single step

This is why we want to be careful about how we design our rules: *we don't want to get too caught up on which rule to apply*

Example

$$\frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{ add-left}$$

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

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

$(\text{sub } 10 \ (\text{add} \ (\text{add } 1 \ 2) \ (\text{add } 2 \ 3))) \longrightarrow ???$

$$\frac{(+ \ 1 \ 2) \longrightarrow 3}{\text{add-ok}}$$

$$\frac{(+ \ (+ \ 1 \ 2) \ (+ \ 2 \ 3)) \longrightarrow (+ \ 3 \ (+ \ 2 \ 3))}{\text{add-left}} \quad \text{sub-right}$$

$$(- \ 10 \ (+ \ (+ \ 1 \ 2) \ (+ \ 2 \ 3))) \longrightarrow (- \ 10 \ (+ \ 3 \ (+ \ 2 \ 3)))$$

Practice Problem

$$\begin{array}{c} \frac{e_1 \longrightarrow e'_1}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e'_1 \ e_2)} \text{add-left} \qquad \frac{e_2 \longrightarrow e'_2}{(\text{add } e_1 \ e_2) \longrightarrow (\text{add } e_1 \ e'_2)} \text{add-right} \\[10pt] \frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{add } n_1 \ n_2) \longrightarrow n_1 + n_2} \text{add-ok} \\[10pt] \frac{e_1 \longrightarrow e'_1}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e'_1 \ e_2)} \text{sub-left} \qquad \frac{e_2 \longrightarrow e'_2}{(\text{sub } e_1 \ e_2) \longrightarrow (\text{sub } e_1 \ e'_2)} \text{sub-right} \\[10pt] \frac{n_1 \text{ is a number} \quad n_2 \text{ is a number}}{(\text{sub } n_1 \ n_2) \longrightarrow n_1 - n_2} \text{sub-ok} \end{array}$$

$(\text{sub } 10 \ (\text{add } 3 \ (\text{add } 2 \ 3))) \longrightarrow (\text{sub } 10 \ (\text{add } 3 \ 5))$

Give a derivation of the above reduction

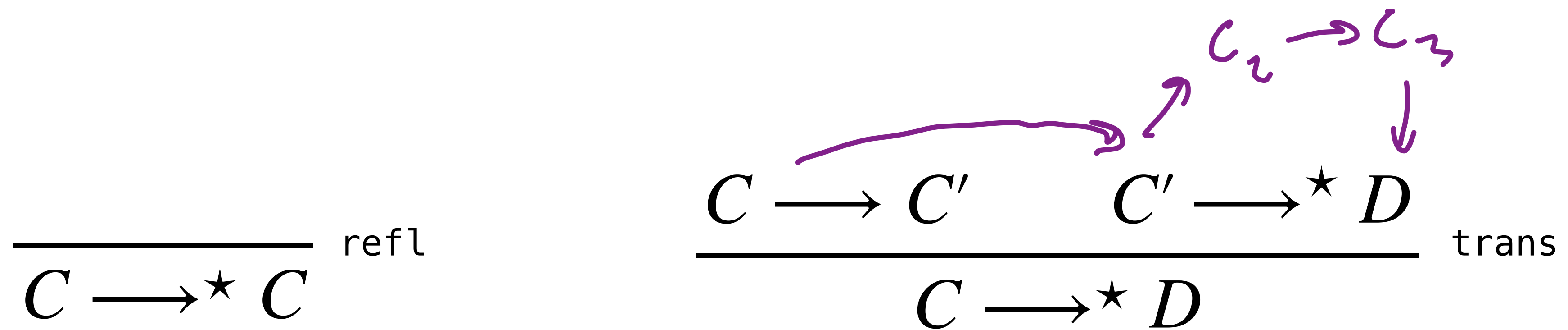
Answer

(add 2 3) → 5 *add-ok*

(add 3 (add 2 3) → (add 3 5)) *add-right*

(sub 10 (add 3 (add 2 3))) → (sub 10 (add 3 5)) *sub-right*

Multi-Step Reduction Relation



Given any single-step reduction relation, we can derive the **multi-step reduction relation**:

- » Every configuration reduces to itself **(reflexivity)**
- » Every \longrightarrow^* reduction can be extended by a single step **(transitivity)**

Two Questions (Again)

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

- » Show that $C \longrightarrow^* C'$
- » Given C , determine a configuration C' such that $C \longrightarrow^* C'$ and C' cannot be reduced

Two Questions (Again)

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

» Show that $C \longrightarrow^* C'$

» Given C , determine a configuration C' such that $C \longrightarrow^* C'$ and C' cannot be reduced

How To: Derivations of Multi-Step Reductions

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2`
want to show

» Derive all necessary single-step evaluations

How To: Derivations of Multi-Step Reductions

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2`
want to show

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))` (we did this)

» Derive all necessary single-step evaluations

How To: Derivations of Multi-Step Reductions

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2`
want to show

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))` (we did this)

`sub 10 (add 3 (add 2 3)) \longrightarrow sub 10 (add 3 5)` (you did this)

» Derive all necessary single-step evaluations

How To: Derivations of Multi-Step Reductions

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2`
want to show

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))` (we did this)

`sub 10 (add 3 (add 2 3)) \longrightarrow sub 10 (add 3 5)` (you did this)

`sub 10 (add 3 5) \longrightarrow sub 10 8` (exercise)

» Derive all necessary single-step evaluations

How To: Derivations of Multi-Step Reductions

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2`
want to show

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))` (we did this)

`sub 10 (add 3 (add 2 3)) \longrightarrow sub 10 (add 3 5)` (you did this)

`sub 10 (add 3 5) \longrightarrow sub 10 8` (exercise)

`sub 10 8 \longrightarrow 2` (easy)

» Derive all necessary single-step evaluations

How To: Derivations of Multi-Step Reductions

`sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2`

- » Derive all necessary single-step evaluations
- » Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

$$\frac{\begin{array}{c} \text{(we did this)} \\ \vdots \\ s\ 10\ (a\ (a\ 1\ 2)\ (a\ 2\ 3)) \longrightarrow s\ 10\ (a\ 3\ (a\ 2\ 3)) \end{array} \quad s\ 10\ (a\ 3\ (a\ 2\ 3)) \longrightarrow^* 2}{\text{sub}\ 10\ (\text{add}\ (\text{add}\ 1\ 2)\ (\text{add}\ 2\ 3)) \longrightarrow^* 2} \text{trans}$$

- » Derive all necessary single-step evaluations
- » Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

$$\begin{array}{c}
 \text{(we did this)} \\
 \vdots \\
 \text{s 10 (a (a 1 2) (a 2 3))} \longrightarrow \text{s 10 (a 3 (a 2 3))} \quad \text{s 10 (a 3 (a 2 3))} \longrightarrow^* 2 \\
 \hline
 \text{sub 10 (add (add 1 2) (add 2 3))} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(you did this)} \\
 \vdots \\
 \text{s 10 (a 3 (a 2 3))} \longrightarrow \text{s 10 (a 3 5)} \quad \text{s 10 (a 3 5)} \longrightarrow^* 2 \\
 \hline
 \text{s 10 (a 3 (a 2 3))} \longrightarrow^* 2
 \end{array}
 \quad
 \text{trans}$$

- » Derive all necessary single-step evaluations
- » Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

$$\begin{array}{c}
 \text{(we did this)} \\
 \vdots \\
 \text{s 10 (a (a 1 2) (a 2 3))} \longrightarrow \text{s 10 (a 3 (a 2 3))} \quad \text{s 10 (a 3 (a 2 3))} \longrightarrow^* 2 \\
 \hline
 \text{sub 10 (add (add 1 2) (add 2 3))} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(you did this)} \\
 \vdots \\
 \text{s 10 (a 3 (a 2 3))} \longrightarrow \text{s 10 (a 3 5)} \quad \text{s 10 (a 3 5)} \longrightarrow^* 2 \\
 \hline
 \text{s 10 (a 3 5)} \longrightarrow \text{s 10 8} \quad \text{s 10 8} \longrightarrow^* 2 \\
 \hline
 \text{s 10 (a 3 5)} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(an exercise)} \\
 \vdots \\
 \text{s 10 (a 3 5)} \longrightarrow \text{s 10 8} \quad \text{s 10 8} \longrightarrow^* 2 \\
 \hline
 \text{s 10 (a 3 5)} \longrightarrow^* 2
 \end{array}
 \quad
 \text{trans}$$

- » Derive all necessary single-step evaluations
- » Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

$$\begin{array}{c}
 \text{(we did this)} \\
 \vdots \\
 \text{s 10 (a (a 1 2) (a 2 3))} \longrightarrow \text{s 10 (a 3 (a 2 3))} \quad \text{s 10 (a 3 (a 2 3))} \longrightarrow^* 2 \\
 \hline
 \text{sub 10 (add (add 1 2) (add 2 3))} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(you did this)} \\
 \vdots \\
 \text{s 10 (a 3 (a 2 3))} \longrightarrow \text{s 10 (a 3 5)} \quad \text{s 10 (a 3 5)} \longrightarrow^* 2 \\
 \hline
 \text{s 10 (a 3 5)} \longrightarrow \text{s 10 8} \quad \text{s 10 8} \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow 2 \quad 2 \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(an exercise)} \\
 \vdots \\
 \text{s 10 8} \longrightarrow 2 \quad 2 \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(easy)} \\
 \vdots \\
 \text{s 10 8} \longrightarrow 2 \quad 2 \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}
 \quad
 \text{trans}$$

- » Derive all necessary single-step evaluations
- » Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

$$\begin{array}{c}
 \text{(we did this)} \\
 \vdots \\
 \text{s 10 (a (a 1 2) (a 2 3))} \longrightarrow \text{s 10 (a 3 (a 2 3))} \quad \text{s 10 (a 3 (a 2 3))} \longrightarrow^* 2 \\
 \hline
 \text{sub 10 (add (add 1 2) (add 2 3))} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(you did this)} \\
 \vdots \\
 \text{s 10 (a 3 (a 2 3))} \longrightarrow \text{s 10 (a 3 5)} \quad \text{s 10 (a 3 5)} \longrightarrow^* 2 \\
 \hline
 \text{s 10 (a 3 5)} \longrightarrow \text{s 10 8} \quad \text{s 10 8} \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow 2 \quad 2 \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(an exercise)} \\
 \vdots \\
 \text{s 10 8} \longrightarrow 2 \quad 2 \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{(easy)} \\
 \vdots \\
 \text{s 10 8} \longrightarrow 2 \quad 2 \longrightarrow^* 2 \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{refl} \\
 \hline
 \text{2} \longrightarrow^* 2
 \end{array}
 \quad
 \begin{array}{c}
 \text{trans} \\
 \hline
 \text{s 10 8} \longrightarrow^* 2
 \end{array}$$

- » Derive all necessary single-step evaluations
- » Combine them with the **transitivity rule**

Two Questions (Again)

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

- » Show that $C \longrightarrow^* C'$
- » Given C , determine a configuration C' such that $C \longrightarrow^* C'$ and C' cannot be further reduced

Two Questions (Again)

Once we have an operational semantics, there are **two questions** we can ask (as PL designers and on the final exam):

» Show that $C \longrightarrow^* C'$

» Given C , determine a configuration C' such that $C \longrightarrow^* C'$ and C' cannot be further reduced

How To: Evaluation

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^{\star} ??

want to show

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow ??

If our rules are well defined, then should be easy:

Solve this single-step evaluation problem until you reach a configuration that cannot be further reduced

How To: Evaluation

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^{\star} ??

want to show

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))

sub 10 (add 3 (add 2 3)) \longrightarrow ??

If our rules are well defined, then should be easy:

Solve this single-step evaluation problem until you reach a configuration that cannot be further reduced

How To: Evaluation

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^{\star} ??

want to show

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))

sub 10 (add 3 (add 2 3)) \longrightarrow sub 10 (add 3 5)

sub 10 (add 3 5) \longrightarrow ??

If our rules are well defined, then should be easy:

Solve this single-step evaluation problem until you reach a configuration that cannot be further reduced

How To: Evaluation

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^{\star} ??

want to show

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))

sub 10 (add 3 (add 2 3)) \longrightarrow sub 10 (add 3 5)

sub 10 (add 3 5) \longrightarrow sub 10 8

sub 10 8 \longrightarrow ??

If our rules are well defined, then should be easy:

Solve this single-step evaluation problem until you reach a configuration that cannot be further reduced

How To: Evaluation

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow^* 2
want to show

sub 10 (add (add 1 2) (add 2 3)) \longrightarrow sub 10 (add 3 (add 2 3))

sub 10 (add 3 (add 2 3)) \longrightarrow sub 10 (add 3 5)

sub 10 (add 3 5) \longrightarrow sub 10 8

sub 10 8 \longrightarrow 2

If our rules are well defined, then should be easy:

Solve this single-step evaluation problem until you reach a configuration that cannot be further reduced

When are we done?

When are we done?

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

When are we done?

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

» **value:** we reach the end of our computation and the value of our program

`(fun x -> x) (2 + 3) →★ 5`

When are we done?

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

» **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

`(fun x -> x) (2 + 3) →★ 5`

`y (fun x -> x) ↯`

When are we done?

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

» **value:** we reach the end of our computation and the value of our program

$(\text{fun } x \rightarrow x) (2 + 3) \rightarrow^* 5$

» **stuck:** we reach an expression that cannot be reduced, but that is not a value

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

» **diverge:** the computation never reaches a point where the expression is not reducible

$(\text{fun } x \rightarrow x x) (\text{fun } x \rightarrow x x) \rightarrow^* (\text{fun } x \rightarrow x x) (\text{fun } x \rightarrow x x)$

moving onto big-step...

Big-Step Semantics

$(\text{sub } 10 (\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3))) \Downarrow 2$

Big-Step Semantics

$(\text{sub } 10 (\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3))) \Downarrow 2$

Big-step semantics deals only with a program and its value

Big-Step Semantics

$(\text{sub } 10 (\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3))) \Downarrow 2$

Big-step semantics deals only with a program and its value

Notation: We write $e \Downarrow v$ to mean that e evaluates to the value v

Big-Step Semantics

$(\text{sub } 10 (\text{add } (\text{add } 1 \ 2) (\text{add } 2 \ 3))) \Downarrow 2$

Big-step semantics deals only with a program and its value

Notation: We write $e \Downarrow v$ to mean that e evaluates to the value v

This is what we've been doing in this course so far

Example

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

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

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

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

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'll remove these side conditions once we have type-checking

Practice Problem

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

Write the rule for eq

Answer

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

Relation to Small-Step

$$e \longrightarrow^{\star} v \qquad \approx \qquad e \Downarrow v$$

The big-step relation "**cuts out the middle steps**" of a small-step relation

This means fewer and clearer rules, but less fine-grain control of the evaluation sequence

Note: We can't always have both small-step and big-step!

Order of Evaluation

.....order of evaluation.....▶

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

With small-step semantics, we can choose the order of evaluations based on the rules

With big-step semantics, we can't because our relation only deals with the *final* value, nothing intermediate

We will take the order of operations to be from left to right

Summary

big-step

$$e \Downarrow v$$

*e evaluates
to v*

single-step

$$e \longrightarrow e'$$

*e reduces to e'
in a single step*

multi-step

$$e \longrightarrow^{\star} e'$$

*e reduces to e'
in many steps*