Formal Semantics

Principles of Programming Languages
Lecture 15

CAS CS 320

Practice Problem

```
<s> ::= A <s> B | <a>
<a> ::= A <a> | A <b>
<b> ::= B <b> | B
```

Show that the above grammar is ambiguous.

Answer

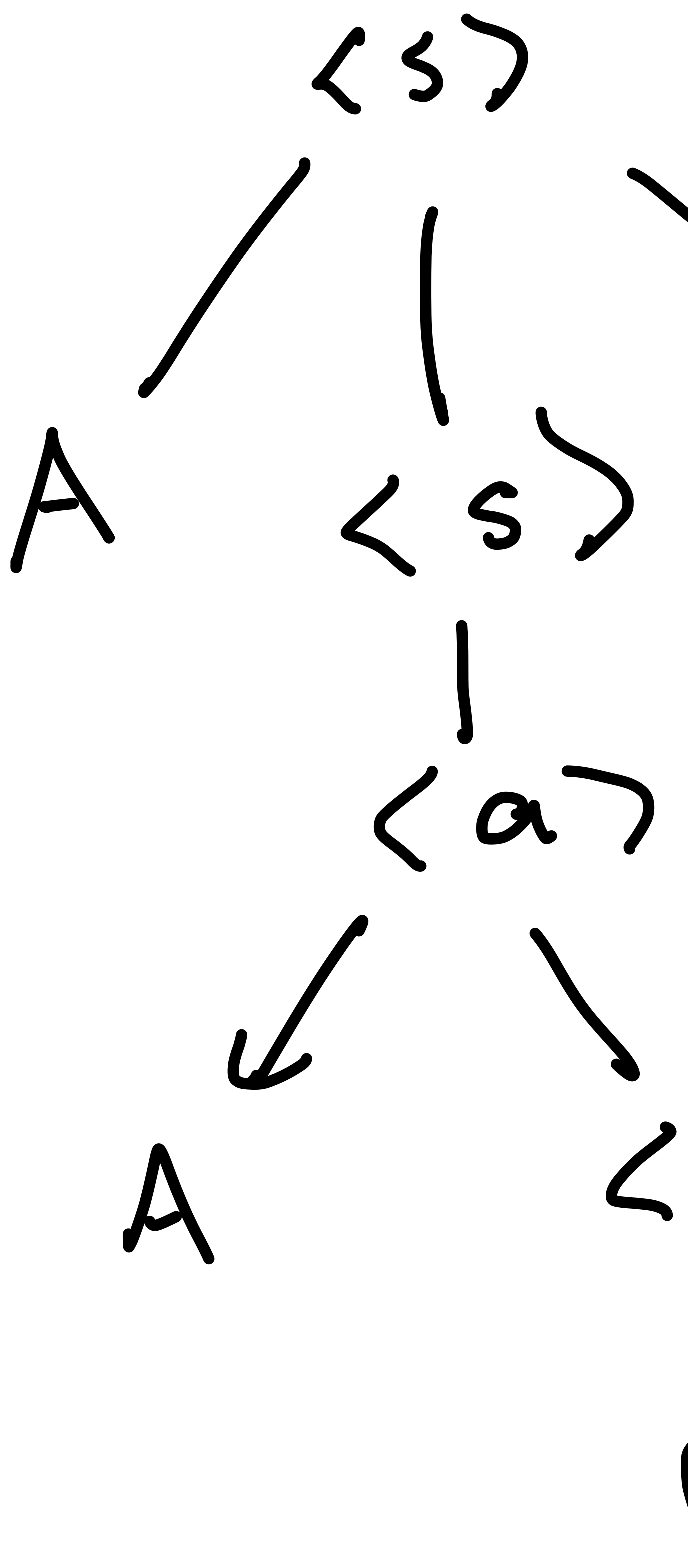
<s> ::= A <s> B | <a>
<a> ::= A <a> | A <b>
<b> ::= B <b> | B

<s>
A <s> B
A <a> B
A A <b> B
A A B B

<s>
<a>
A <a>
A A <b>
A A B <b>
A A B B

A A B

$\boxed{\text{A A B B}}$



Outline

Discuss formal semantics in general

Look at small-step and big-step semantics with some examples

Learning Objectives

- Determine the value of the expression e according to a given operational semantics
- Describe the difference between big step and small step semantics
- Derive $e \longrightarrow^{\star} e'$ or $e \Downarrow v$
- Determine the order of evaluation given by a set of semantics rules (when possible)
- What does this program print?

Introduction

A Thought Experiment

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

Bash

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

Python

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

OCaml

A Thought Experiment

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

Bash

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

Python

```ocaml
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?*

A Thought Experiment

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

Bash

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

Python

```ocaml
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

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

Bash

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

Python

```ocaml
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

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

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

```
2 + 3
```

```
5
```

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?

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

↓

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

↓

```
2 + 3
```

↓

```
5
```

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

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

↓

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

↓

```
2 + 3
```

↓

```
5
```

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*

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

$$1 + 12 - 2 = 11$$

$1 + 1 \Downarrow 2$

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*

$$1 + 2 * 3 + 4 = 11$$
$$1 + 12 - 2 = 11$$

This course

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

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

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

$$\dfrac{2 \Downarrow 2 \qquad \dfrac{3 \Downarrow 3 \qquad \dfrac{2 \Downarrow 2 \qquad 3 \Downarrow 3}{\texttt{2 + 3} \Downarrow 5}}{\texttt{let y = 3 in 2 + y} \Downarrow 5}}{\texttt{let x = 2 in let y = 3 in x + y} \Downarrow 5}$$

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

$$\dfrac{2 \Downarrow 2 \qquad \dfrac{3 \Downarrow 3 \qquad \dfrac{2 \Downarrow 2 \qquad 3 \Downarrow 3}{\texttt{2 + 3} \Downarrow 5}}{\texttt{let y = 3 in 2 + y} \Downarrow 5}}{\texttt{let x = 2 in let y = 3 in x + y} \Downarrow 5}$$

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

```
% ocaml silly.py
File "./silly.py", 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.py
File "./silly.py", 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

let x = 2+.3.

Type checking

Static semantics refers to the meaning given to a program *before* it is evaluated

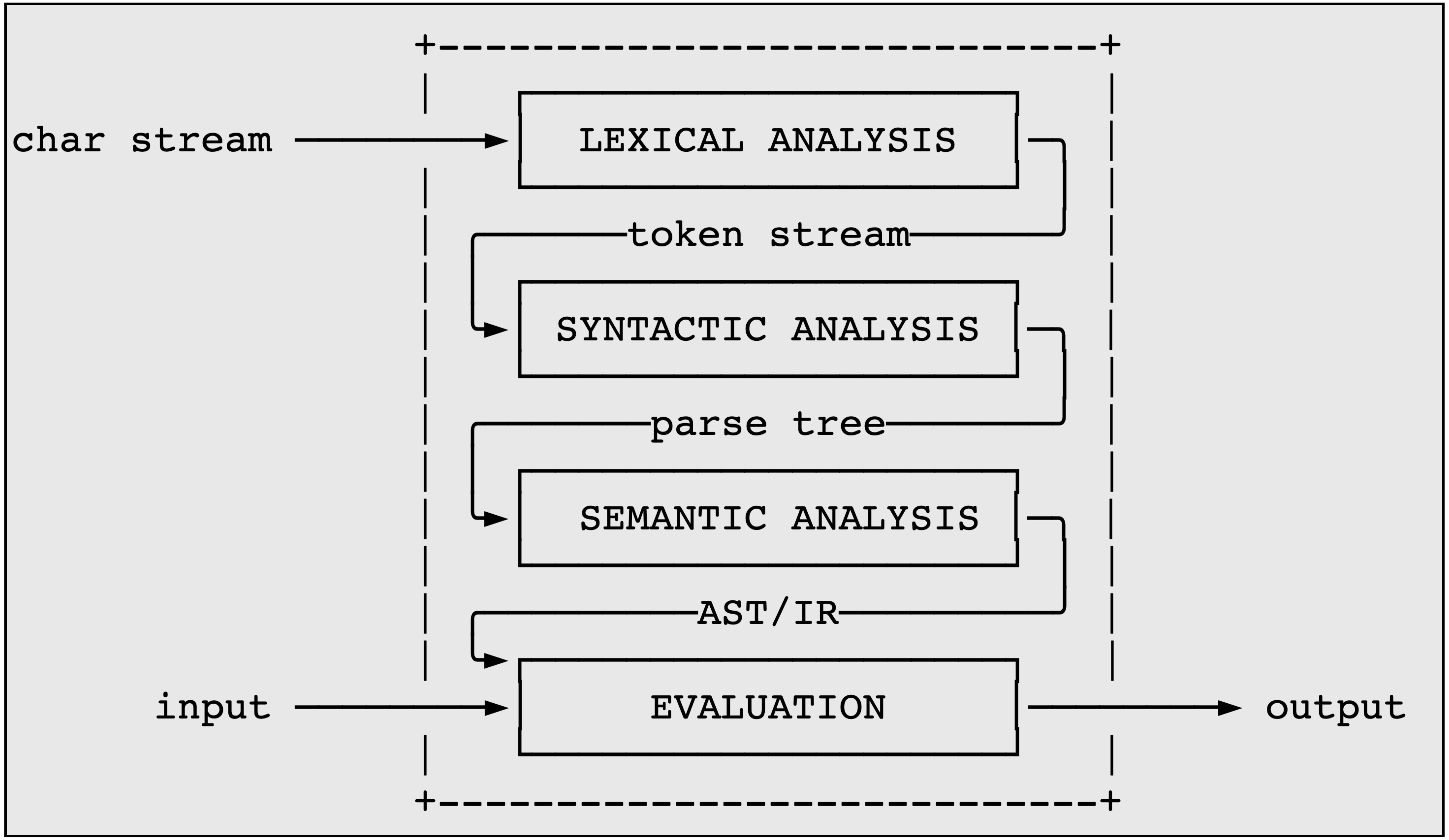
```
% ocaml silly.py
File "./silly.py", line 1, characters 8-9:
1 | let x = 2 +. 3.
            ^
Error: This expression has type int but an expression was
       expected of type
         float
  Hint: Did you mean `2.'?
```

Evaluation

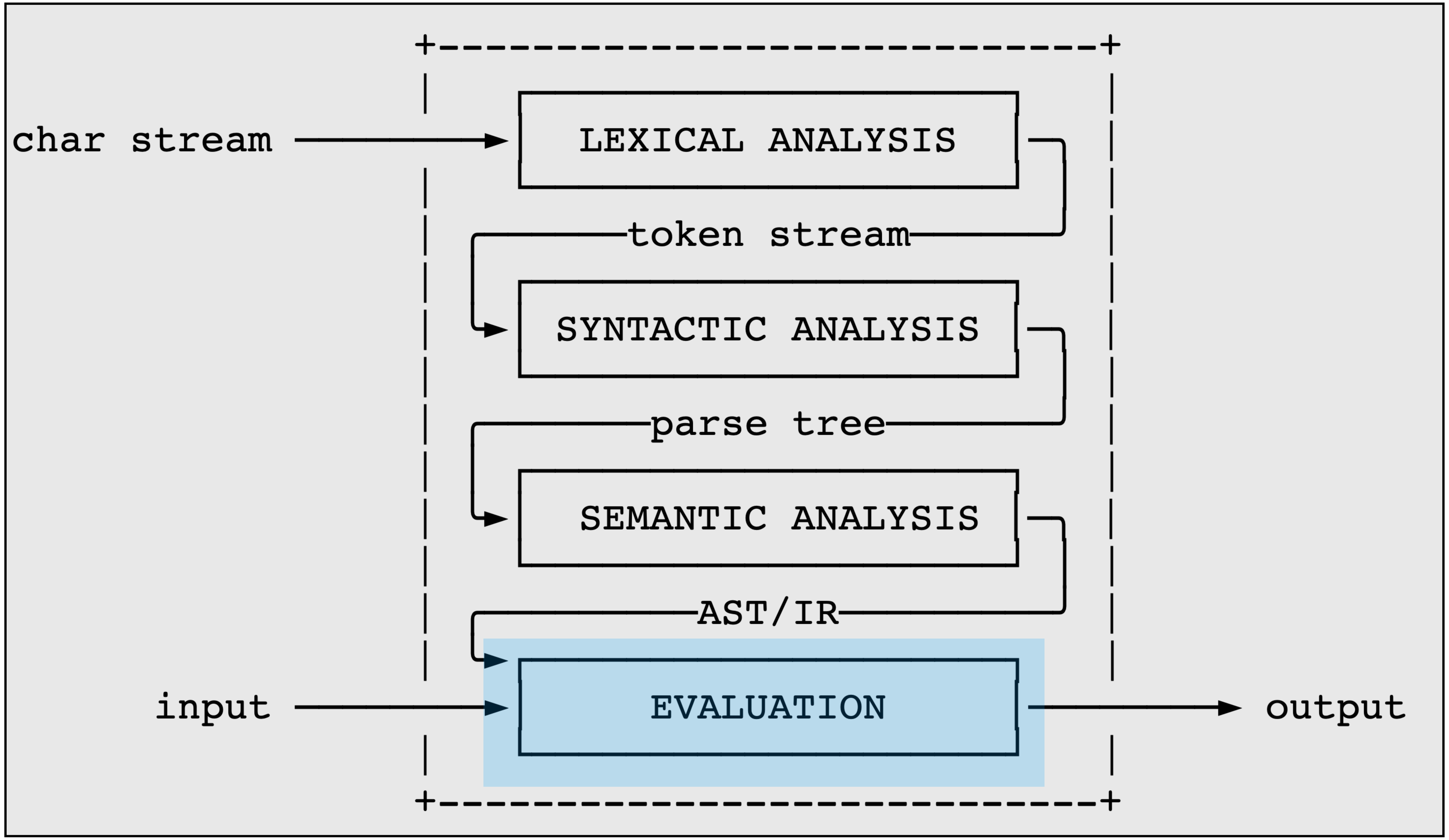
Dynamic semantics refers to the behavior of a program *during* evaluation

```
utop # let x = 2 + 3;;
val x : int = 5
```

Recall: The Picture

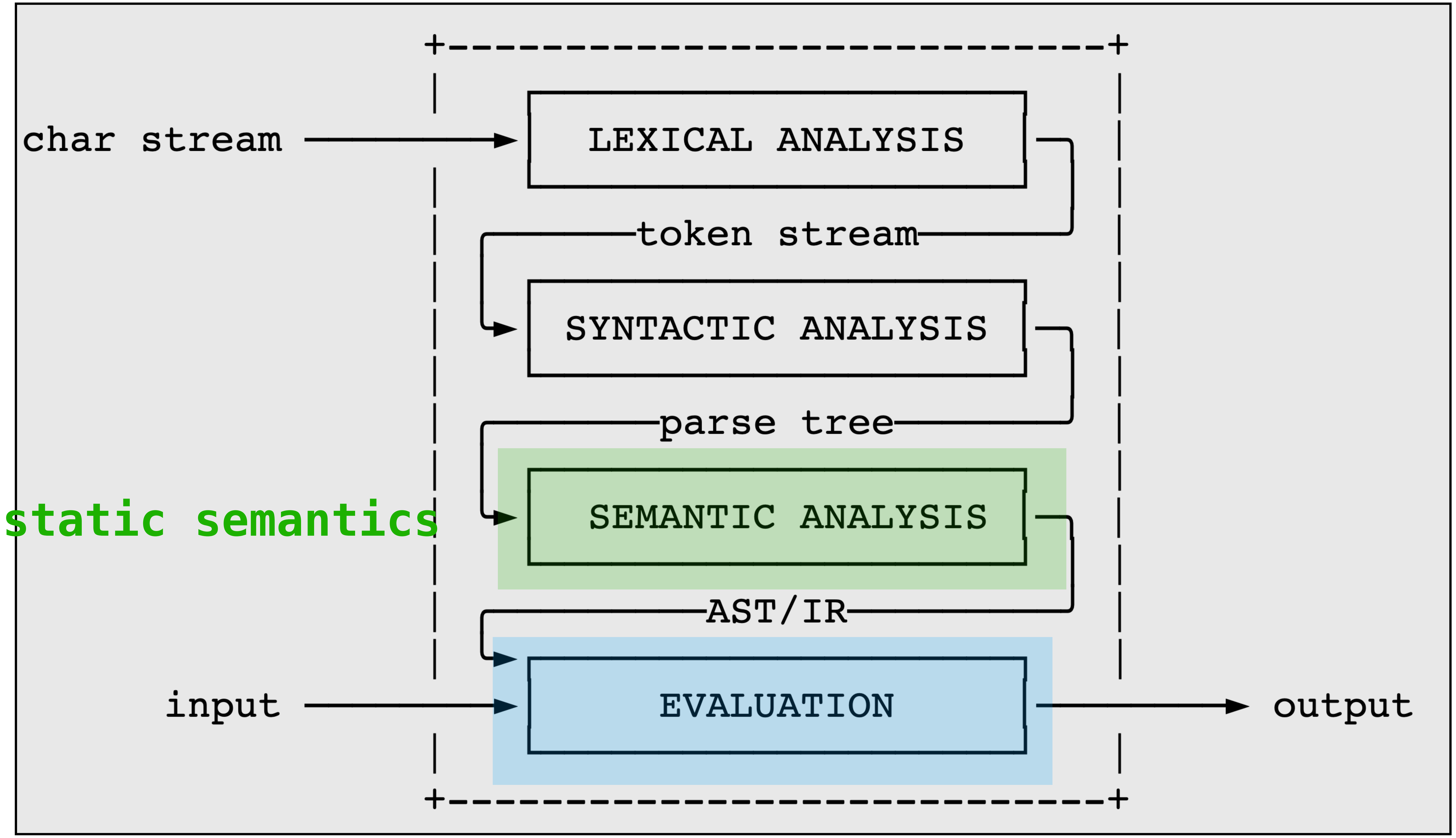


Recall: The Picture



dynamic semantics (this week + next week)

Recall: The Picture



dynamic semantics (this week + next week)

Operational Semantics

Inference Rules

$$\frac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}} \; \overset{\text{name}}{\texttt{red-rule}}$$

Inference Rules

$$\frac{\overset{\texttt{premise}}{P_1} \qquad \overset{\texttt{premise}}{P_2} \qquad \dots \qquad \overset{\texttt{premise}}{P_k}}{\underset{\texttt{conclusion}}{C}}\ \overset{\texttt{name}}{\texttt{red-rule}}$$

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

Inference Rules

$$\frac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}}\ \overset{\text{name}}{\texttt{red-rule}}$$

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**

Inference Rules

$$\dfrac{\overset{\text{premise}}{P_1} \qquad \overset{\text{premise}}{P_2} \qquad \cdots \qquad \overset{\text{premise}}{P_k}}{\underset{\text{conclusion}}{C}}\ \overset{\text{name}}{\texttt{red-rule}}$$

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**

Premises which are not of the same form as the conclusion are called **side-conditions**

Example

premise

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{add}\ e_1\ e_2) \longrightarrow (\texttt{add}\ e_1'\ e_2)}\ \texttt{add-left}$$

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)
```

Example

premise

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

conclusion

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

<u>Example Programs:</u>

```
(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* add $e_1\ e_2$ *reduces to* add $e_1'\ e_2$ *in one step*

Another Example

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

If n_1 and n_2 are numbers then $(\texttt{add}\ n_1\ n_2)$ 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

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

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

$$(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})$$

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

configuration — transformed configuration

program — transformed program

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

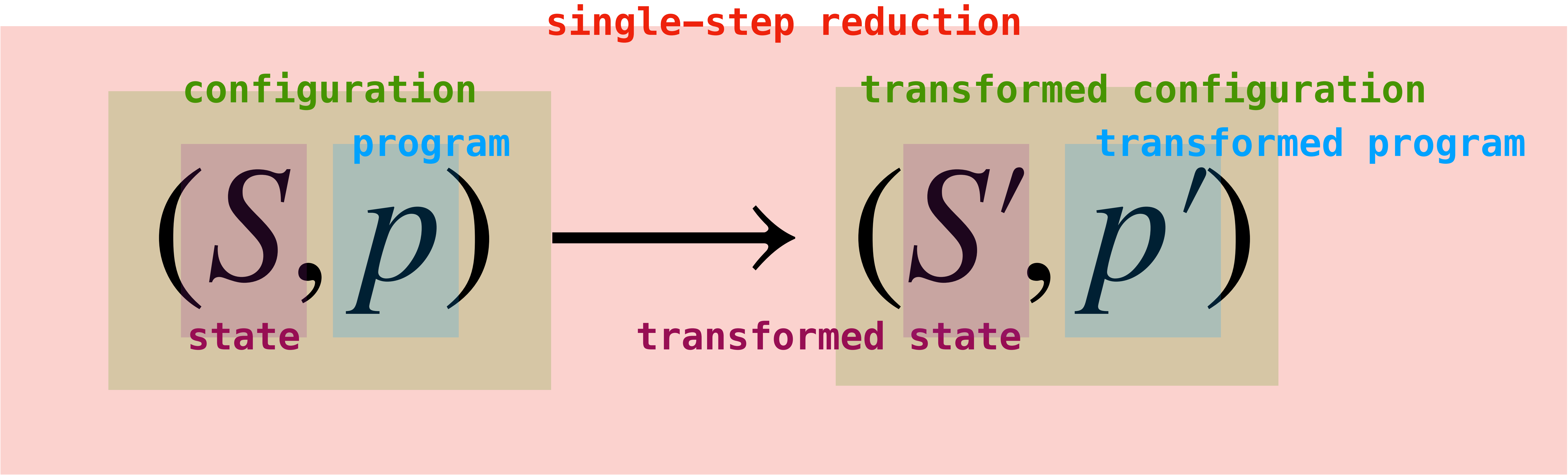
state — transformed state

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

$$\underbrace{(\varnothing}_{\texttt{state}}\,,\ \overbrace{10 \times (2 + 3)}^{\texttt{program}}) \longrightarrow (\varnothing\,,\ 10 \times 5) \longrightarrow (\varnothing\,,\ 50)$$

<u>State:</u> none

<u>Program:</u> arithmetic expression

Example: (Fragment of) OCaml

program

(∅ , `let x = 3 in if x > 10 then 4 else 5`) ⟶ (∅ , `if 3 > 10 then 4 else 5`)

state

⟶ (∅ , `if false then 4 else 5`)

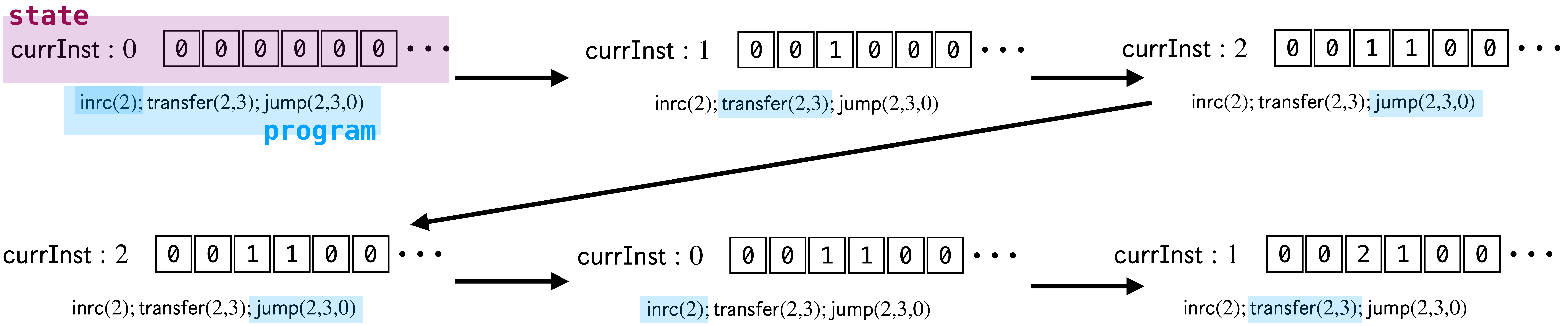
⟶ (∅ , `5`)

State: none

Program: OCaml expression

For purely functional languages
there is no state

Example: Unlimited Register Machines



<u>State:</u> (current instruction pointer) + (collection of number registers)

<u>Program:</u> sequence of commands for updating registers values and current instruction

Example: Stack-Oriented Language

state program

(∅ , push 2; push 3; add) ⟶

(2 :: ∅, push 3; add) ⟶

(3 :: 2 :: ∅, add) ⟶

(5 :: ∅, ϵ)

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

Program: sequence of commands for manipulating the stack

High-Level

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

High-Level

$$(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})$$

When we define the small-step semantics of a programming language, we need to define two things:

High-Level

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

When we define the small-step semantics of a programming language, we need to define two things:

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

High-Level

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

When we define the small-step semantics of a programming language, we need to define two things:

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

» What **rules** describe how to transform configurations?

High-Level

$$\underbrace{(S}_{\text{state}},\overbrace{p}^{\text{program}}) \longrightarrow (\underbrace{S'}_{\text{transformed state}},\overbrace{p'}^{\text{transformed program}})$$

When we define the small-step semantics of a programming language, we need to define two things:

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

» What **rules** describe how to transform configurations?

(we'll elide the state for most of this course)

Example

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

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{add}\ e_1\ e_2) \longrightarrow (\texttt{add}\ e_1'\ e_2)}\ \texttt{add-left} \qquad \frac{e_2 \longrightarrow e_2'}{(\texttt{add}\ e_1\ e_2) \longrightarrow (\texttt{add}\ e_1\ e_2')}\ \texttt{add-right}$$

$$\frac{n_1\ \texttt{is a number} \qquad n_2\ \texttt{is a number}}{(\texttt{add}\ n_1\ n_2) \longrightarrow n_1 \boxed{+} n_2}\ \texttt{add-ok}$$

$$\frac{2 \in \mathbb{Z} \qquad 3 \in \mathbb{Z}}{(\text{add } 2\ 3) \to 5}$$

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{sub}\ e_1\ e_2) \longrightarrow (\texttt{sub}\ e_1'\ e_2)}\ \texttt{sub-left} \qquad \frac{e_2 \longrightarrow e_2'}{(\texttt{sub}\ e_1\ e_2) \longrightarrow (\texttt{sub}\ e_1\ e_2')}\ \texttt{sub-right}$$

$$\frac{n_1\ \texttt{is a number} \qquad n_2\ \texttt{is a number}}{(\texttt{sub}\ n_1\ n_2) \longrightarrow n_1 \boxed{-} n_2}\ \texttt{sub-ok}$$

$$\frac{(\text{add } 2\ 3) \to 5}{\text{add } (\text{add } 2\ 3)\ 5 \to \text{add } 5\ 5}$$

Reduction is a Relation

$$\begin{tikzcd}
 & (1+2)+(2+3) \arrow[dl] \arrow[dr] & \\
(1+2)+5 \arrow[dr] & & 3+(2+3) \arrow[dl] \\
 & 8 &
\end{tikzcd}$$

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

This means there may be multiple choices of reductions

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

Reduction is a Relation

$$\dfrac{\texttt{add}\ 1\ 2 \longrightarrow 3}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{add-left}$$

$$\dfrac{\texttt{add}\ 2\ 3 \longrightarrow 5}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ (\texttt{add}\ 1\ 2)\ 5)}\ \texttt{add-right}$$

Reduction is a Relation

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

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

There are two reductions from $(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))$ in our current rule set.

Reduction is a Relation

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

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

There are two reductions from $(\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3))$ in our current rule set.

We can avoid this by *breaking symmetry*. We will enforce that the right argument can 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>  ::= ...
```

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{add}\ e_1\ e_2) \longrightarrow (\texttt{add}\ e_1'\ e_2)}\ \texttt{add-left}$$

$$\frac{v\ \texttt{is a number} \qquad e_2 \longrightarrow e_2'}{(\texttt{add}\ v\ e_2) \longrightarrow (\texttt{add}\ v\ e_2')}\ \texttt{add-right}$$

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

Enforcing an Evaluation Order

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

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

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

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>  ::= ...
```

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{eq}\ e_1\ e_2) \longrightarrow (\texttt{eq}\ e_1'\ e_2)}$$

$$\frac{v\ \texttt{is a num or bool} \qquad e_2 \longrightarrow e_2'}{(\texttt{eq}\ v\ e_2) \longrightarrow (\texttt{eq}\ v\ e_2')}$$

$$\frac{b_1\ \texttt{is a bool} \qquad b_2\ \texttt{is a bool}}{(\texttt{eq}\ b_1\ b_2) \longrightarrow b_1 = b_2}$$

$$\frac{n_1\ \texttt{is a num} \qquad n_2\ \texttt{is a num}}{(\texttt{eq}\ n_1\ n_2) \longrightarrow n_1 = n_2}$$

Answer

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

$$\frac{e_1 \longrightarrow e_1'}{(\texttt{eq}\ e_1\ e_2) \longrightarrow (\texttt{eq}\ e_1'\ e_2)}$$

$$\frac{v \texttt{ is a num or bool} \qquad e_2 \longrightarrow e_2'}{(\texttt{eq}\ v\ e_2) \longrightarrow (\texttt{eq}\ v\ e_2')}$$

$$\frac{b_1 \texttt{ is a bool} \qquad b_2 \texttt{ is a bool}}{(\texttt{eq}\ b_1\ b_2) \longrightarrow b_1 = b_2}$$

$$\frac{n_1 \texttt{ is a num} \qquad n_2 \texttt{ is a num}}{(\texttt{eq}\ n_1\ n_2) \longrightarrow n_1 = n_2}$$

Looks a lot like pattern matching.

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).

Derivations

$$\dfrac{\texttt{10 is a number} \qquad \dfrac{\dfrac{\texttt{1 is a number} \qquad \texttt{2 is a number}}{(\texttt{add}\ 1\ 2) \longrightarrow 3}\ \texttt{add-ok}}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{add-left}}{(\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))) \longrightarrow (\texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)))}\ \texttt{sub-right}$$

Derivations

$$\dfrac{10 \texttt{ is a number} \qquad \dfrac{\dfrac{1 \texttt{ is a number} \qquad 2 \texttt{ is a number}}{(\texttt{add}\ 1\ 2) \longrightarrow 3}\ \texttt{add-ok}}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{add-left}}{(\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))) \longrightarrow (\texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)))}\ \texttt{sub-right}$$

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

Derivations

$$\dfrac{10 \texttt{ is a number} \qquad \dfrac{\dfrac{1 \texttt{ is a number} \qquad 2 \texttt{ is a number}}{(\mathsf{add}\ 1\ 2) \longrightarrow 3}\ \texttt{add-ok}}{(\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3))}\ \texttt{add-left}}{(\mathsf{sub}\ 10\ (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3))) \longrightarrow (\mathsf{sub}\ 10\ (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3)))}\ \texttt{sub-right}$$

Definition (Informal): 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.

How To: Building Derivations

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

How To: Building Derivations

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

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

How To: Building Derivations

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

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.

How To: Building Derivations

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

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.

How To: Building Derivations

$$\frac{10 \ \texttt{is a number} \qquad (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}{\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{sub-right}$$

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.

How To: Building Derivations

$$\frac{\texttt{10 is a number} \qquad (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3))}{\mathsf{sub}\ 10\ (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow \mathsf{sub}\ 10\ (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3))}\ \texttt{sub-right}$$

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.

How To: Building Derivations

$$\frac{\texttt{10 is a number} \qquad (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}{\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{sub-right}$$

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.

How To: Building Derivations

$$\dfrac{\texttt{10 is a number} \qquad \dfrac{(\texttt{add}\ 1\ 2) \longrightarrow 3}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{add-left}}{\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{sub-right}$$

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.

How To: Building Derivations

$$\dfrac{10 \texttt{ is a number} \qquad \dfrac{\dfrac{1 \texttt{ is a number} \qquad 2 \texttt{ is a number}}{(\texttt{add } 1\ 2) \longrightarrow 3}\ \texttt{add-ok}}{(\texttt{add } (\texttt{add } 1\ 2)\ (\texttt{add } 2\ 3)) \longrightarrow (\texttt{add } 3\ (\texttt{add } 2\ 3))}\ \texttt{add-left}}{\texttt{sub } 10\ (\texttt{add } (\texttt{add } 1\ 2)\ (\texttt{add } 2\ 3)) \longrightarrow \texttt{sub } 10\ (\texttt{add } 3\ (\texttt{add } 2\ 3))}\ \texttt{sub-right}$$

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.

How To: Building Derivations

$$\dfrac{\texttt{10 is a number} \qquad \dfrac{\dfrac{\texttt{1 is a number} \qquad \texttt{2 is a number}}{(\texttt{add}\ 1\ 2) \longrightarrow 3}\ \texttt{add-ok}}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{add-left}}{\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))}\ \texttt{sub-right}$$

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

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

Single-Step Evaluation

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

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

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

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.*

How To: Performing Single-Step Evaluation

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

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

How To: Performing Single-Step Evaluation

$$\overbrace{(\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)))}^{\texttt{sub}\ n\ e} \longrightarrow ??$$

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

How To: Performing Single-Step Evaluation

$$\frac{10 \ \texttt{is a number} \qquad (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow ??}{(\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))) \longrightarrow (\texttt{sub}\ 10\ ??)}\ \texttt{sub-right}$$

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

How To: Performing Single-Step Evaluation

$$\frac{10 \ \texttt{is a number} \qquad \overset{\texttt{add}\ e_1\ e_2}{(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))} \longrightarrow ??}{(\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))) \longrightarrow (\texttt{sub}\ 10\ ??)}\ \texttt{sub-right}$$

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

How To: Performing Single-Step Evaluation

$$\dfrac{10\ \texttt{is a number} \qquad \dfrac{\mathsf{add}\ 1\ 2 \longrightarrow ??}{(\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow (\mathsf{add}\ ??\ (\mathsf{add}\ 2\ 3))}\ \texttt{add-left}}{(\mathsf{sub}\ 10\ (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3))) \longrightarrow (\mathsf{sub}\ 10\ (\mathsf{add}\ ??\ (\mathsf{add}\ 2\ 3)))}\ \texttt{sub-right}$$

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

How To: Performing Single-Step Evaluation

$$
\dfrac{\texttt{10 is a number} \qquad \dfrac{\overset{\mathsf{add}\ n_1\ n_2}{\mathsf{add}\ 1\ 2} \longrightarrow\ ??}{(\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow (\mathsf{add}\ ??\ (\mathsf{add}\ 2\ 3))}\ \texttt{add-left}}{(\mathsf{sub}\ 10\ (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3))) \longrightarrow (\mathsf{sub}\ 10\ (\mathsf{add}\ ??\ (\mathsf{add}\ 2\ 3)))}\ \texttt{sub-right}
$$

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

How To: Performing Single-Step Evaluation

$$\dfrac{10 \texttt{ is a number} \qquad \dfrac{\dfrac{1 \texttt{ is a number} \qquad 2 \texttt{ is a number}}{\mathsf{add}\ 1\ 2 \longrightarrow 3}\ \texttt{add-ok}}{(\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3))}\ \texttt{add-left}}{(\mathsf{sub}\ 10\ (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3))) \longrightarrow (\mathsf{sub}\ 10\ (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3)))}\ \texttt{sub-right}$$

We can perform a single evaluation step by again, build derivations from the ground up.

If we've designed our rules well (e.g., by enforcing evaluation order) there should always be a rule to use.

Practice Problem

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

Give a derivation of the above reduction

Answer

$$\dfrac{10 \in \mathbb{Z} \qquad \dfrac{3 \in \mathbb{Z} \qquad \dfrac{2 \in \mathbb{Z} \quad 3 \in \mathbb{Z} \quad 2+3=5}{(\text{add } 2\ 3) \to 5}\ \text{add-ok}}{\text{add } 3\ (\text{add } 2\ 3) \to (\text{add } 3\ 5)}\ \text{add-right}}{(\text{sub } 10\ (\text{add } 3\ (\text{add } 2\ 3))) \longrightarrow (\text{sub } 10\ (\text{add } 3\ 5))}\ \text{sub-right}$$

Multi-Step Reduction Relation

$C \to C' \to C'' \to C''' \to \cdots \to D$

$$\frac{}{C \longrightarrow^\star C}\ \texttt{refl} \qquad\qquad \frac{C \longrightarrow C' \qquad C' \longrightarrow^\star D}{C \longrightarrow^\star D}\ \texttt{trans}$$

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

» Every configuration reduces to itself **(reflexivity)**

» Every $\longrightarrow^\star$ 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 $C \longrightarrow^\star C'$.
- Given C, determine a configuration C' such that $C \longrightarrow^\star C'$ **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 $C \longrightarrow^{\star} C'$.
- Given C, determine a configuration C' such that $C \longrightarrow^{\star} C'$ **and** C' **cannot be reduced.**

How To: Derivations of Multi-Step Reductions

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

want to show

» Derive all necessary single-step evaluations.

How To: Derivations of Multi-Step Reductions

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

want to show

$$\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))$$ (we did this)

» Derive all necessary single-step evaluations.

How To: Derivations of Multi-Step Reductions

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

want to show

$\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))$ `(we did this)`

$\texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ 5)$ `(you did this)`

» Derive all necessary single-step evaluations.

How To: Derivations of Multi-Step Reductions

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

want to show

$\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))$ (we did this)

$\texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ 5)$ (you did this)

$\texttt{sub}\ 10\ (\texttt{add}\ 3\ 5) \longrightarrow \texttt{sub}\ 10\ 8$ (exercise)

» Derive all necessary single-step evaluations.

How To: Derivations of Multi-Step Reductions

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

`want to show`

$\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3))$ `(we did this)`

$\texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ 5)$ `(you did this)`

$\texttt{sub}\ 10\ (\texttt{add}\ 3\ 5) \longrightarrow \texttt{sub}\ 10\ 8$ `(exercise)`

$\texttt{sub}\ 10\ 8 \longrightarrow 2$ `(easy)`

» Derive all necessary single-step evaluations.

How To: Derivations of Multi-Step Reductions

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

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

How To: Derivations of Multi-Step Reductions

$$\dfrac{\begin{array}{c}\texttt{(we did this)}\\ \vdots\\ \texttt{s}\ 10\ (\texttt{a}\ (\texttt{a}\ 1\ 2)\ (\texttt{a}\ 2\ 3)) \longrightarrow \texttt{s}\ 10\ (\texttt{a}\ 3\ (\texttt{a}\ 2\ 3))\end{array} \qquad \texttt{s}\ 10\ (\texttt{a}\ 3\ (\texttt{a}\ 2\ 3)) \longrightarrow^{\star} 2}{\texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow^{\star} 2}\ \texttt{trans}$$

» Derive all necessary single-step evaluations

» Combine them with the **transitivity rule.**

How To: Derivations of Multi-Step Reductions

$$
\dfrac{
  \dfrac{\texttt{(we did this)}}{\vdots}
  \atop
  \texttt{s}\ 10\ (\texttt{a}\ (\texttt{a}\ 1\ 2)\ (\texttt{a}\ 2\ 3)) \longrightarrow \texttt{s}\ 10\ (\texttt{a}\ 3\ (\texttt{a}\ 2\ 3))
  \qquad
  \dfrac{
    \dfrac{\texttt{(you did this)}}{\vdots}
    \atop
    \texttt{s}\ 10\ (\texttt{a}\ 3\ (\texttt{a}\ 2\ 3)) \longrightarrow \texttt{s}\ 10\ (\texttt{a}\ 3\ 5)
    \qquad
    \texttt{s}\ 10\ (\texttt{a}\ 3\ 5) \longrightarrow^{\star} 2
  }{
    \texttt{s}\ 10\ (\texttt{a}\ 3\ (\texttt{a}\ 2\ 3)) \longrightarrow^{\star} 2
  }\,\texttt{trans}
}{
  \texttt{sub}\ 10\ (\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3)) \longrightarrow^{\star} 2
}\,\texttt{trans}
$$

» Derive all necessary single-step evaluations

» Combine them with the **transitivity rule.**

How To: Derivations of Multi-Step Reductions

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

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

How To: Derivations of Multi-Step Reductions

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

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

How To: Derivations of Multi-Step Reductions

$$
\dfrac{
\dfrac{\texttt{(we did this)}}{\vdots} \quad
\mathtt{s}\ 10\ (\mathtt{a}\ (\mathtt{a}\ 1\ 2)\ (\mathtt{a}\ 2\ 3)) \longrightarrow \mathtt{s}\ 10\ (\mathtt{a}\ 3\ (\mathtt{a}\ 2\ 3))
\qquad
\dfrac{
\dfrac{\texttt{(you did this)}}{\vdots} \quad
\mathtt{s}\ 10\ (\mathtt{a}\ 3\ (\mathtt{a}\ 2\ 3)) \longrightarrow \mathtt{s}\ 10\ (\mathtt{a}\ 3\ 5)
\qquad
\dfrac{
\dfrac{\texttt{(an exercise)}}{\vdots} \quad
\mathtt{s}\ 10\ (\mathtt{a}\ 3\ 5) \longrightarrow \mathtt{s}\ 10\ 8
\qquad
\dfrac{
\dfrac{\texttt{(easy)}}{\vdots} \quad
\mathtt{s}\ 10\ 8 \longrightarrow 2
\qquad
\dfrac{}{2 \longrightarrow^{\star} 2}\,\texttt{refl}
}{\mathtt{s}\ 10\ 8 \longrightarrow^{\star} 2}\,\texttt{trans}
}{\mathtt{s}\ 10\ (\mathtt{a}\ 3\ 5) \longrightarrow^{\star} 2}\,\texttt{trans}
}{\mathtt{s}\ 10\ (\mathtt{a}\ 3\ (\mathtt{a}\ 2\ 3)) \longrightarrow^{\star} 2}\,\texttt{trans}
}{\mathtt{sub}\ 10\ (\mathtt{add}\ (\mathtt{add}\ 1\ 2)\ (\mathtt{add}\ 2\ 3)) \longrightarrow^{\star} 2}\,\texttt{trans}
$$

- 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 show that it holds).

How To: Evaluation

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

want to show

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

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

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

want to show

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

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

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

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

want to show

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

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

$$\mathsf{sub}\ 10\ (\mathsf{add}\ 3\ 5) \longrightarrow\ ??$$

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

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

want to show

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

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

$$\texttt{sub}\ 10\ (\texttt{add}\ 3\ 5) \longrightarrow \texttt{sub}\ 10\ 8$$

$$\texttt{sub}\ 10\ 8 \longrightarrow\ ??$$

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

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

want to show

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

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

$$\texttt{sub}\ 10\ (\texttt{add}\ 3\ 5) \longrightarrow \texttt{sub}\ 10\ 8$$

$$\texttt{sub}\ 10\ 8 \longrightarrow 2$$

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

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

$\ldots \to (\text{add } 5\ 3) \to \boxed{8}$

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

$\ldots \to (\text{add true } 3) \not\to$

there are no stuck terms with type checking!

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

$e_1 \to e_2 \to e_3 \to e_1 \to e_2 \to \ldots$

moving onto big-step...

Big-Step Semantics

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

Big-Step Semantics

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

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

Big-Step Semantics

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

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

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

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

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

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

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

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

Example

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

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

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

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

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

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

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

What is a value?

$$e \Downarrow \underbrace{\boxed{v}}_{\texttt{value}}$$

What is a value?

$$e \Downarrow \underbrace{\boxed{v}}_{\text{value}}$$

Anything we want it to be

What is a value?

$$e \Downarrow \boxed{v}$$

value

Anything we want it to be

Often, as in small-step semantics, a **value** is a special kind of *expression*

What is a value?

$$e \Downarrow \boxed{v}^{\,\text{value}}$$

Anything we want it to be

Often, as in small-step semantics, a **value** is a special kind of *expression*

But we get to **choose** what our values are (we will usually define them separately as an ADT)

What is a value?

$$e \Downarrow \boxed{v}^{\texttt{value}}$$

Anything we want it to be

Often, as in small-step semantics, a **value** is a special kind of *expression*

But we get to **choose** what our values are (we will usually define them separately as an ADT)

This will turn out to be very useful for mini-project 2

Taking Stock

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

demo

(how does this look in code)