Formal Semantics

Concepts of Programming Languages
Lecture 14

CAS CS 320

Outline

Discuss formal semantics in general

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

demo

(finish up the Menhir demo)

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
```

$\downarrow$

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

$\downarrow$

```
2 + 3
```

$\downarrow$

```
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*

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

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

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

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.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.'?
```

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

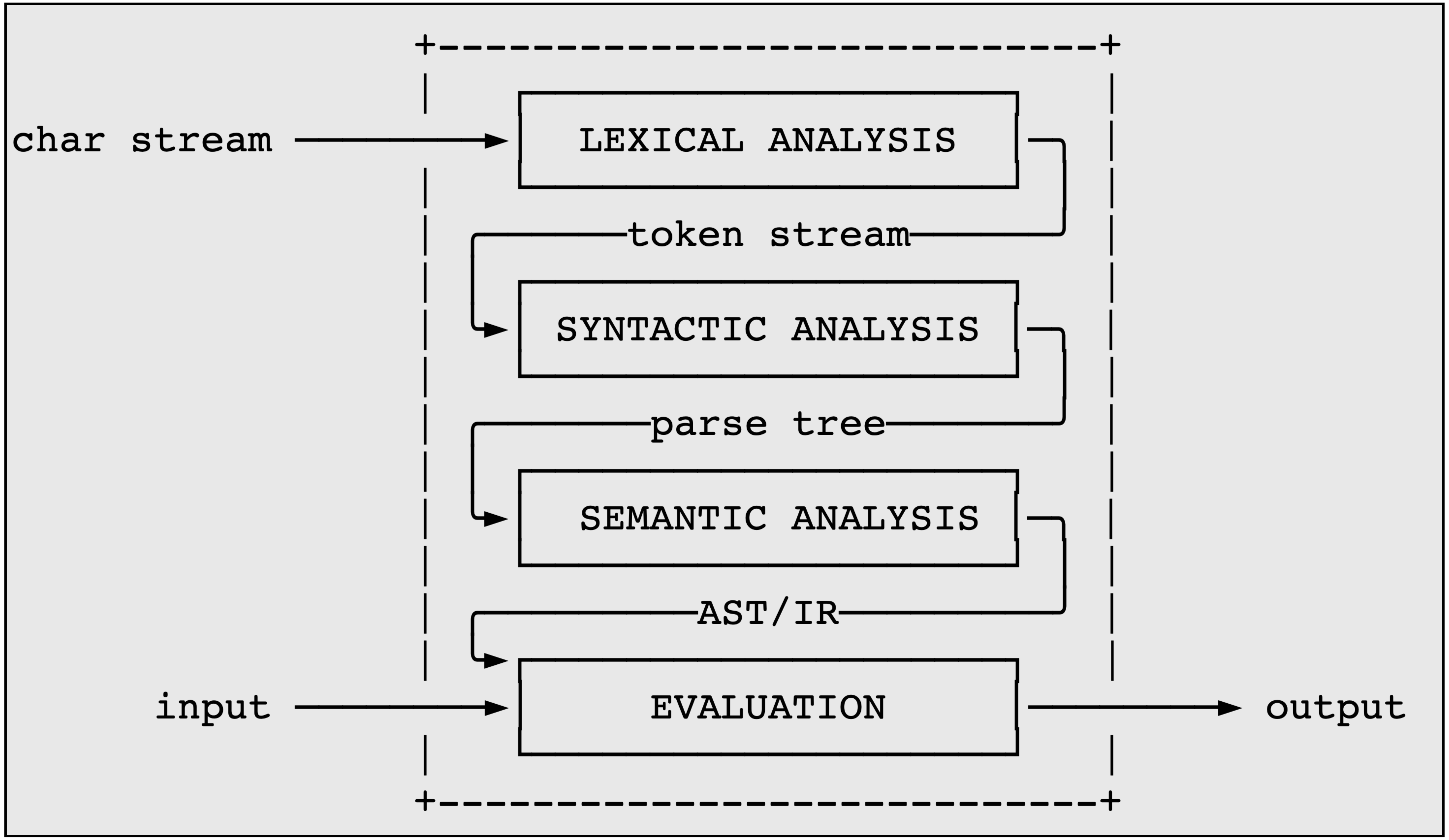
```
% ocaml silly.ml
File "./silly.ml", line 1, characters 8-9:
1 | let x = 2 +. 3.
            ^
Error: This expression has type int but an expression was
       expected of type
         float
  Hint: Did you mean `2.'?
```

Evaluation

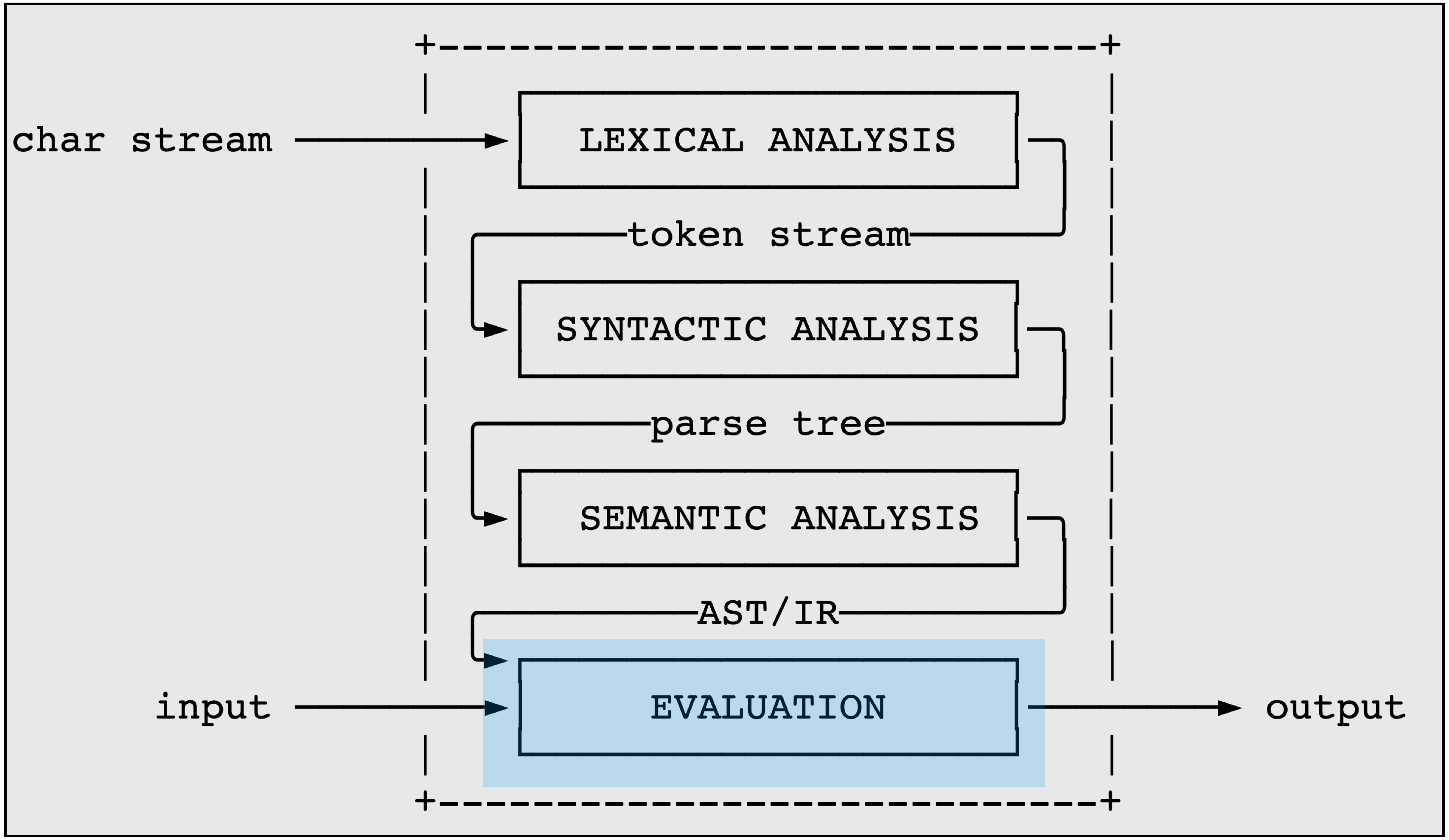
Dynamic semantics
refers to the behavior of a program *during* evaluation

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

Recall: The Picture

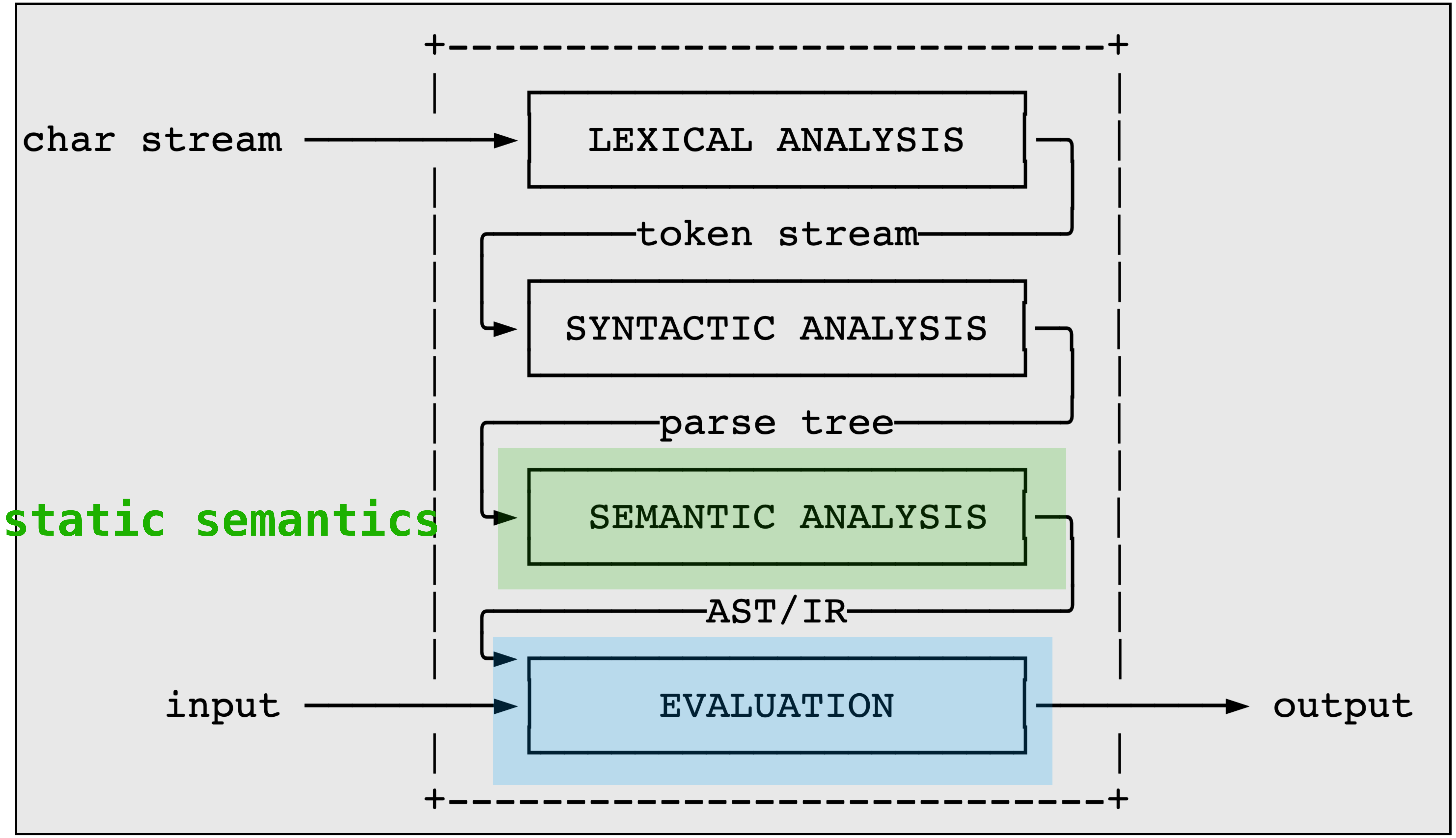


Recall: The Picture



dynamic semantics (this week + next week)

Recall: The Picture



dynamic semantics (this week + next week)

Operational Semantics

Recall: Inference Rules

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

Recall: 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**

Recall: Inference Rules

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

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**

Example

premise

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

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

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

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)
```

If e_1 *reduces to* e_1' *in one step, then* $\mathrm{add}\ e_1\ e_2$ *reduces to* $\mathrm{add}\ e_1'\ e_2$ *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

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

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

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

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

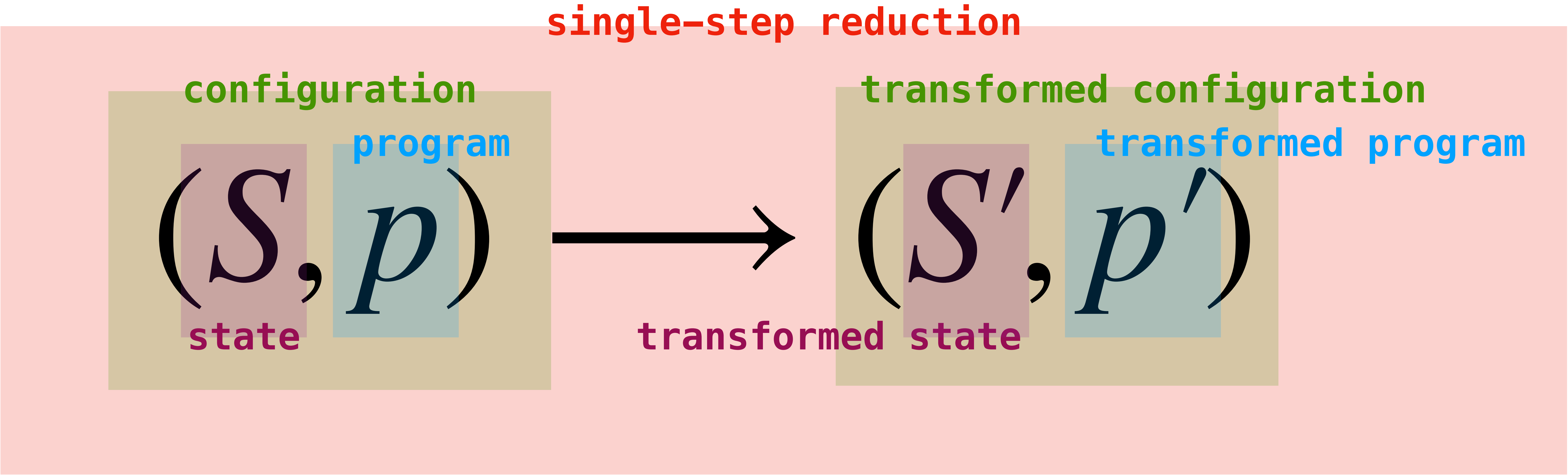
$$\overbrace{(\underbrace{S}_{\text{state}}, \overbrace{p}^{\text{program}})}^{\text{configuration}} \longrightarrow \overbrace{(\underbrace{S'}_{\text{transformed state}}, \overbrace{p'}^{\text{transformed program}})}^{\text{transformed configuration}}$$

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

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

State: none

Program: arithmetic expression

Example: (Fragment of) OCaml

program

$(\varnothing$, `let x = 3 in if x > 10 then 4 else 5`$)$ $\longrightarrow (\varnothing$, `if 3 > 10 then 4 else 5`$)$

state

$\longrightarrow (\varnothing$, `if false then 4 else 5`$)$

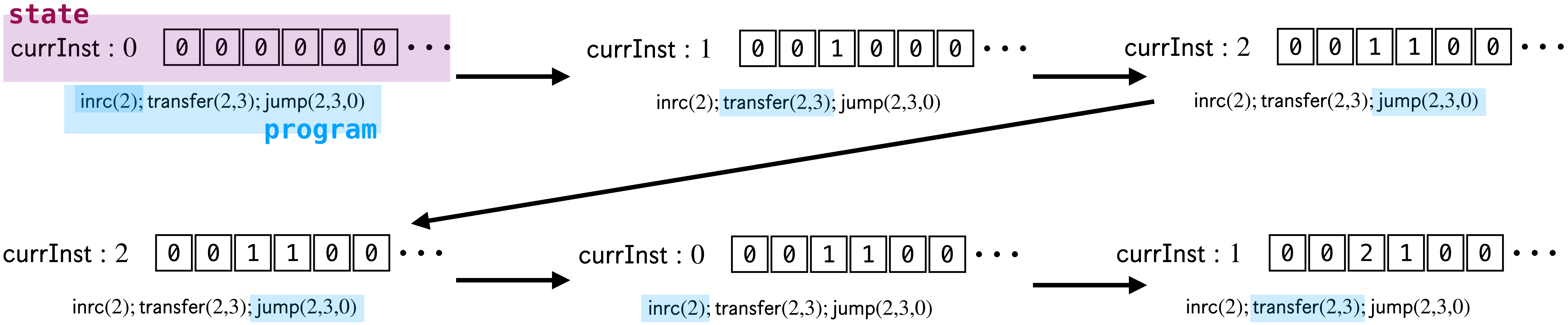
$\longrightarrow (\varnothing$, `5`$)$

<u>State:</u> none

<u>Program:</u> 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

$$\underbrace{(\varnothing}_{\text{state}}, \underbrace{\texttt{push 2; push 3; add})}_{\text{program}} \longrightarrow$$

$$(2 :: \varnothing, \texttt{push 3; add}) \longrightarrow$$

$$(3 :: 2 :: \varnothing, \texttt{add}) \longrightarrow$$

$$(5 :: \varnothing, \epsilon)$$

<u>State:</u> stack (i.e., list) of values

<u>Program:</u> sequence of commands for manipulating the stack

High-Level

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

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 PL, we need to define two things:

High-Level

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

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

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

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 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>  ::= ...
```

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

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

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

Example

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

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

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

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

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

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

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

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>  ::= ...
```

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

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

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

Example

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

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

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

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

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

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

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

Reduction is a Relation

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

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

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

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

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

Reduction is a Relation

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

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

There are two reductions from $(\textsf{add}\ (\textsf{add}\ 1\ 2)\ (\textsf{add}\ 2\ 3))$ 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>  ::= ...
```

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

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

Recall: Derivations

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

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

Recall: Derivations

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

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

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

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

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

Recall: 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

Recall: 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

Recall: 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

Recall: Building Derivations

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

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

Recall: Building Derivations

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

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

Recall: Building Derivations

$$\dfrac{\dfrac{(\mathsf{add}\ 1\ 2) \longrightarrow 3}{(\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3))}\ \texttt{add-left}}{\mathsf{sub}\ 10\ (\mathsf{add}\ (\mathsf{add}\ 1\ 2)\ (\mathsf{add}\ 2\ 3)) \longrightarrow \mathsf{sub}\ 10\ (\mathsf{add}\ 3\ (\mathsf{add}\ 2\ 3))}\ \texttt{sub-right}$$

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

Recall: Building Derivations

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

We can 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

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

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

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*

Example

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

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

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

side cond.

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

$$\dfrac{\dfrac{\dfrac{}{(\text{add } 1\ 2) \to 3}\ \text{add-ok}}{(\text{add } (\text{add } 1\ 2)\ (\text{add } 2\ 3)) \to \text{add } 3\ (\text{add } 2\ 3)}\ \text{add-l}}{\text{sub } 10\ (\text{add } (\text{add } 1\ 2)\ (\text{add } 2\ 3)) \to \text{sub } 10\ (\text{add } 3\ (\text{add } 2\ 3))}\ \text{sub-right}$$

Practice Problem

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

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

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

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

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

Give a derivation of the above reduction

Answer

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

Multi-Step Reduction Relation

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

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

(we did this)

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

» Derive all necessary single-step evaluations

» Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

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

» Derive all necessary single-step evaluations

» Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

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

» Derive all necessary single-step evaluations

» Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

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

» Derive all necessary single-step evaluations

» Combine them with the **transitivity rule**

How To: Derivations of Multi-Step Reductions

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

- 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 $C \longrightarrow^{\star} C'$
- 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: Evaluation

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

want to show

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

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

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

want to show

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

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

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

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

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

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

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

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

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** "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

true + "two"

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

let rec f x = f x in f 0

$\Downarrow$

f 0 => f 0 ...

moving onto big-step...

Big-Step Semantics

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

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
```

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

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

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

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

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