Administrivia

Homework 8 has been updated. It is due on 4/6 *Saturday by 11:59PM*

An early reminder that the final exam for this course is *Wednesday 5/8, 3–5PM in STO B50*

Project 1 will be assigned this week

No discussion sections this week, but we will have office hours during the discussion sections in the same locations (except for Section A6)

Formal Semantics II: Applying Rules

Principles of Programming Languages
Lecture 18

CAS CS 320

Objectives

Recap ***operational semantics***.

Practice building reduction ***derivations*** for a collection of small examples.

See how this translates to ***OCaml code***.

Keywords

operational semantics

configuration

abstract machine

reduction rule

derivation

single-step reduction

multi-step reduction

stack-oriented language

Recap: Operational Semantics

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)

demo

(Python vs. Bash vs. OCaml)

High-Level

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

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level

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

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level

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

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level

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

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

High-Level

single-step reduction

configuration

program

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

state

transformed configuration

transformed program

transformed state

Definition (Informal): A **program** is a thing which is transformed (reduced) by *evaluation* and may update some kind of state

Evaluation is the process of *reducing* a program repeatedly until it is no longer possible to do so

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

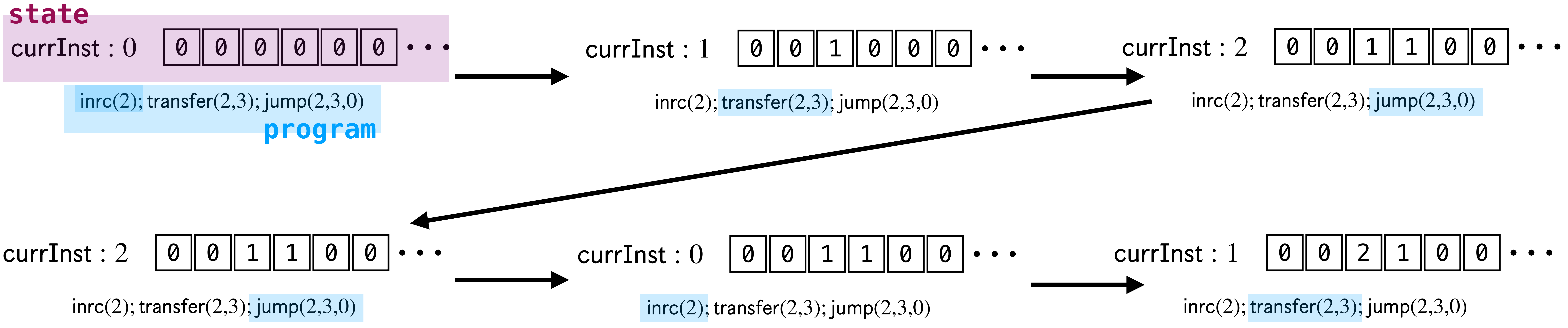
$$\underbrace{(\varnothing}_{\texttt{state}}, \overbrace{\texttt{let x = 3 in if x > 10 then 4 else 5}}^{\texttt{program}}) \longrightarrow (\varnothing, \texttt{if 3 > 10 then 4 else 5})$$
$$\longrightarrow (\varnothing, \texttt{if false then 4 else 5})$$
$$\longrightarrow (\varnothing, \texttt{5})$$

`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

$(\varnothing$, push 2; push 3; add$) \longrightarrow$

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

$(3 :: 2 :: \varnothing$, add$) \longrightarrow$

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

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

Program: sequence of commands for manipulating the stack

Example: Stack-Oriented Language with Trace

state / program

((∅ , ∅), push 2; trace; push 3; add; trace) ⟶

((2 :: ∅, ∅), trace; push 3; add; trace) ⟶

((2 :: ∅, "2" :: ∅), push 3; add; trace) ⟶

((3 :: 2 :: ∅, "2" :: ∅), add; trace) ⟶

((5 :: ∅, "2" :: ∅), trace) ⟶

((5 :: ∅, "5" :: "2" :: ∅), ϵ)

<u>State:</u> stack of values + trace of strings to print

<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

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

When we define the operational semantics of a programming language, we need to define two things:

High-Level

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

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

When we define the operational semantics of a programming language, 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 operational 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

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

When we define the operational 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?

Expressing Possible Reductions

$$
\begin{array}{lll}
(\varnothing\ ,\ 1+2) \to (\varnothing\ ,\ 3) & (\varnothing\ ,\ 1+(1+2)) \to (\varnothing\ ,\ 1+3) & \\
(\varnothing\ ,\ 1+3) \to (\varnothing\ ,\ 4) & (\varnothing\ ,\ 1+(1+3)) \to (\varnothing\ ,\ 1+4) & \cdots \\
(\varnothing\ ,\ 1+4) \to (\varnothing\ ,\ 5) & (\varnothing\ ,\ 1+(1+4)) \to (\varnothing\ ,\ 1+5) & \\
\vdots & \vdots & \ddots
\end{array}
$$

One Approach: *list them all.*

This is fine for small programming languages, but doesn't scale well.

We'd rather be able to give a concise description of the shapes of possible reductions.

Reduction Rules

premise premise premise

$$\frac{P_1 \qquad P_2 \qquad \cdots \qquad P_k}{C \to C'}\ \texttt{red-rule}$$

name

conclusion

Reduction Rules

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

Then general form of a reduction rule has a collection of **premises** and a **conclusion**, which is a shape of a reduction

Reduction Rules

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

Then general form of a reduction rule has a collection of **premises** and a **conclusion,** which is a shape of a reduction

Think of a "shape" like an OCaml "pattern"

Reduction Rules

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

Then general form of a reduction rule has a collection of **premises** and a **conclusion**, which is a shape of a reduction

Think of a "shape" like an OCaml "pattern"

A premise may be another reduction "shape" or a **trivial condition** (we will see examples in the next slide)

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> | ERROR
<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

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

conclusion

```
<expr> ::= ( <op> <expr> <expr> )
         | <bool> | <int> | ERROR
<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*

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> | ERROR
<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* add e_1 e_2 *reduces to* add e_1' e_2 *in one step*

In this case, the premise is another reduction

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 add n_1 n_2 *reduces in one step to* ***the number*** $n_1 + n_2$

In this case, the premises are trivial conditions (it should be easy to determine if something is a number)

Rules for Addition

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

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

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

$$\frac{v\ \texttt{is a Bool or Error}}{\texttt{add}\ v\ e \longrightarrow \texttt{Error}}\ \texttt{add-left-error}$$

$$\frac{v\ \texttt{is a Bool or Error}}{\texttt{add}\ e\ v \longrightarrow \texttt{Error}}\ \texttt{add-right-error}$$

error handling

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

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

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

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

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

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

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

There are two reductions from $(\texttt{add}\ (\texttt{add}\ 1\ 2)\ (\texttt{add}\ 2\ 3))$ 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.

Rules for Addition

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

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

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

$$\frac{v\ \texttt{is a Bool or Error}}{\texttt{add}\ v\ e \longrightarrow \texttt{Error}}\ \texttt{add-left-error}$$

$$\frac{v\ \texttt{is a Bool or Error}}{\texttt{add}\ e\ v \longrightarrow \texttt{Error}}\ \texttt{add-right-error}$$

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

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.

Understanding Check

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

To the best of your ability, write down the reduction rules for ***eq***

When should there be an error?

What order should the arguments be evaluated?

Answer

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

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

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

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

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

$$\frac{b\ \texttt{is a bool} \qquad n\ \texttt{is a num}}{(\texttt{eq}\ b\ n) \longrightarrow \text{ERROR}}$$

$$\frac{n\ \texttt{is a num} \qquad b\ \texttt{is a bool}}{(\texttt{eq}\ n\ b) \longrightarrow \text{ERROR}}$$

Answer

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

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

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

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

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

$$\frac{b\ \texttt{is a bool} \qquad n\ \texttt{is a num}}{(\texttt{eq}\ b\ n) \longrightarrow \mathsf{ERROR}}$$

$$\frac{n\ \texttt{is a num} \qquad b\ \texttt{is a bool}}{(\texttt{eq}\ n\ b) \longrightarrow \mathsf{ERROR}}$$

Looks a lot like pattern matching.

Evaluation

Taking Stock

Taking Stock

Specifying the operational semantics of a programming language means:

Taking Stock

Specifying the operational semantics of a programming language means:

» Deciding on what a **configuration** is (i.e., what **state** there is, if any)

Taking Stock

Specifying the operational semantics of a programming language means:

- Deciding on what a **configuration** is (i.e., what **state** there is, if any)
- Providing all the **reduction rules** for configurations

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

How To: Building Derivations

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

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 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).

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

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

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

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

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

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

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

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

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

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

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

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

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

We can 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.

Understanding Check

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

Give a derivation of the above reduction.

Answer

$$
\dfrac{10\ \texttt{is a number} \qquad \dfrac{3\ \texttt{is a number} \qquad \dfrac{2\ \texttt{is a number} \qquad 3\ \texttt{is a number}}{(\texttt{add}\ 2\ 3) \longrightarrow 5}\ \texttt{add-ok}}{(\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)) \longrightarrow (\texttt{add}\ 3\ 5)}\ \texttt{add-right}}{\texttt{sub}\ 10\ (\texttt{add}\ 3\ (\texttt{add}\ 2\ 3)) \longrightarrow \texttt{sub}\ 10\ (\texttt{add}\ 3\ 5)}\ \texttt{sub-right}
$$

demo

(single-step evaluator in OCaml)
(interlude: recursive parsers)

Multi-Step Reduction Relation

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

Given any single-step (a.k.a. small-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)

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

» 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

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

- 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

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

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

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

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

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

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

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

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

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

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

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

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

demo

(multi–step evaluator in OCaml)

Stack-Oriented Language

Our Toy Language

```
<prog> ::= { <com> }
<com>  ::= Push <int> | Pop | Swap | Add
<int>  ::= ...
```

Our Toy Language

```
<prog> ::= { <com> }
<com>  ::= Push <int> | Pop | Swap | Add
<int>  ::= ...
```

What is our configuration?

Our Toy Language

```
<prog> ::= { <com> }
<com>  ::= Push <int> | Pop | Swap | Add
<int>  ::= ...
```

What is our configuration?

State: A list of integers or **ERROR.**

Our Toy Language

```
<prog> ::= { <com> }
<com>  ::= Push <int> | Pop | Swap | Add
<int>  ::= ...
```

`What is our configuration?`

State: A list of integers or **ERROR**.

Program: A sequence of <com> commands.

The Operational Semantics

The Operational Semantics

$$\frac{}{(S\texttt{, Push n } P) \longrightarrow (n :: S\texttt{, } P)}\ \texttt{push}$$

The Operational Semantics

$$\frac{}{(S,\ \texttt{Push n}\ P) \longrightarrow (n :: S,\ P)}\ \texttt{push}$$

$$\frac{}{(n :: S,\ \texttt{Pop}\ P) \longrightarrow (S,\ P)}\ \texttt{pop-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Pop}\ P) \longrightarrow (\texttt{ERROR},\ \epsilon)}\ \texttt{pop-error}$$

The Operational Semantics

$$\frac{}{(S,\ \texttt{Push n}\ P) \longrightarrow (n :: S,\ P)}\ \texttt{push}$$

$$\frac{}{(n :: S,\ \texttt{Pop}\ P) \longrightarrow (S,\ P)}\ \texttt{pop-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Pop}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{pop-error}$$

$$\frac{}{(m :: n :: S,\ \texttt{Swap}\ P) \longrightarrow (n :: m :: S,\ P)}\ \texttt{swap-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Swap}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{swap-error-0}$$

$$\frac{}{(n :: \varnothing,\ \texttt{Swap}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{swap-error-1}$$

The Operational Semantics

$$\frac{}{(S,\ \texttt{Push n}\ P) \longrightarrow (n :: S,\ P)}\ \texttt{push}$$

$$\frac{}{(n :: S,\ \texttt{Pop}\ P) \longrightarrow (S,\ P)}\ \texttt{pop-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Pop}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{pop-error}$$

$$\frac{}{(m :: n :: S,\ \texttt{Add}\ P) \longrightarrow ((m+n) :: S,\ P)}\ \texttt{add-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Add}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{add-error-0}$$

$$\frac{}{(n :: \varnothing,\ \texttt{Add}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{add-error-1}$$

$$\frac{}{(m :: n :: S,\ \texttt{Swap}\ P) \longrightarrow (n :: m :: S,\ P)}\ \texttt{swap-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Swap}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{swap-error-0}$$

$$\frac{}{(n :: \varnothing,\ \texttt{Swap}\ P) \longrightarrow (\mathsf{ERROR},\ \epsilon)}\ \texttt{swap-error-1}$$

The Operational Semantics

$$\frac{}{(S,\ \texttt{Push n}\ P) \longrightarrow (n :: S,\ P)}\ \texttt{push}$$

$$\frac{}{(n :: S,\ \texttt{Pop}\ P) \longrightarrow (S,\ P)}\ \texttt{pop-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Pop}\ P) \longrightarrow (\texttt{ERROR},\ \epsilon)}\ \texttt{pop-error}$$

$$\frac{}{(m :: n :: S,\ \texttt{Add}\ P) \longrightarrow ((m+n) :: S,\ P)}\ \texttt{add-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Add}\ P) \longrightarrow (\texttt{ERROR},\ \epsilon)}\ \texttt{add-error-0}$$

$$\frac{}{(n :: \varnothing,\ \texttt{Add}\ P) \longrightarrow (\texttt{ERROR},\ \epsilon)}\ \texttt{add-error-1}$$

$$\frac{}{(m :: n :: S,\ \texttt{Swap}\ P) \longrightarrow (n :: m :: S,\ P)}\ \texttt{swap-ok}$$

$$\frac{}{(\varnothing,\ \texttt{Swap}\ P) \longrightarrow (\texttt{ERROR},\ \epsilon)}\ \texttt{swap-error-0}$$

$$\frac{}{(n :: \varnothing,\ \texttt{Swap}\ P) \longrightarrow (\texttt{ERROR},\ \epsilon)}\ \texttt{swap-error-1}$$

Note.

» Each rule has no premise

» Each rule behaves like OCaml pattern-matching

Example: Evaluating a Program

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing, \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 1: Derive all necessary single-step reductions.

Example: Evaluating a Program

(1) $$\dfrac{}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow (1 :: \varnothing\ ,\ \texttt{Push 2 Swap Add})}\ \texttt{push}$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 1: Derive all necessary single-step reductions.

Example: Evaluating a Program

(1)
$$\frac{}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow (1 :: \varnothing\ ,\ \texttt{Push 2 Swap Add})}\ \texttt{push}$$

(2)
$$\frac{}{(1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow (2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add})}\ \texttt{push}$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 1: Derive all necessary single-step reductions.

Example: Evaluating a Program

(1)
$$\frac{}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow (1 :: \varnothing\ ,\ \texttt{Push 2 Swap Add})}\ \texttt{push}$$

(2)
$$\frac{}{(1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow (2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add})}\ \texttt{push}$$

(3)
$$\frac{}{(2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add}) \longrightarrow (1 :: 2 :: \varnothing\ ,\ \texttt{Add})}\ \texttt{swap-ok}$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 1: Derive all necessary single-step reductions.

Example: Evaluating a Program

(1)
$$\frac{}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow (1 :: \varnothing\ ,\ \texttt{Push 2 Swap Add})}\ \texttt{push}$$

(2)
$$\frac{}{(1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow (2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add})}\ \texttt{push}$$

(3)
$$\frac{}{(2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add}) \longrightarrow (1 :: 2 :: \varnothing\ ,\ \texttt{Add})}\ \texttt{swap-ok}$$

(4)
$$\frac{}{(1 :: 2 :: \varnothing\ ,\ \texttt{Add}) \longrightarrow (3 :: \varnothing\ ,\ \epsilon)}\ \texttt{add-ok}$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 1: Derive all necessary single-step reductions.

Example: Evaluating a Program

(1) $$\frac{}{(\varnothing, \texttt{Push 1 Push 2 Swap Add}) \longrightarrow (1 :: \varnothing\ , \texttt{Push 2 Swap Add})}\ \texttt{push}$$

(2) $$\frac{}{(1 :: \varnothing, \texttt{Push 2 Swap Add}) \longrightarrow (2 :: 1 :: \varnothing\ , \texttt{Swap Add})}\ \texttt{push}$$

(3) $$\frac{}{(2 :: 1 :: \varnothing\ , \texttt{Swap Add}) \longrightarrow (1 :: 2 :: \varnothing\ , \texttt{Add})}\ \texttt{swap-ok}$$

(4) $$\frac{}{(1 :: 2 :: \varnothing\ , \texttt{Add}) \longrightarrow (3 :: \varnothing\ , \epsilon)}\ \texttt{add-ok}$$

Again, choosing the right rule is like pattern matching.

Question. Evaluate `Push 1 Push 2 Swap Add` in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing, \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 1: Derive all necessary single-step reductions.

Example: Evaluating a Program

$$(\varnothing, \texttt{Push 1 Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing \ , \ \epsilon)$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing, \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 2: Combine using the transitivity rule.

Example: Evaluating a Program

$$\frac{(1)\quad (1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\ \texttt{trans}$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 2: Combine using the transitivity rule.

Example: Evaluating a Program

$$
\dfrac{(1)\quad \dfrac{(2)\quad (2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}{(1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}
$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 2: Combine using the transitivity rule.

Example: Evaluating a Program

$$\dfrac{(1)\quad (1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow^\star (3 :: \varnothing,\ \epsilon) \qquad \dfrac{(2)\quad (2 :: 1 :: \varnothing,\ \texttt{Swap Add}) \longrightarrow^\star (3 :: \varnothing,\ \epsilon) \qquad \dfrac{(3)\quad (1 :: 2 :: \varnothing,\ \texttt{Add}) \longrightarrow^\star (3 :: \varnothing,\ \epsilon)}{}\,\texttt{trans}}{}\,\texttt{trans}}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow^\star (3 :: \varnothing,\ \epsilon)}\,\texttt{trans}$$

Question. Evaluate Push 1 Push 2 Swap Add in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 2: Combine using the transitivity rule.

Example: Evaluating a Program

$$\dfrac{(1)\quad \dfrac{(2)\quad \dfrac{(3)\quad \dfrac{(4)\quad (3 :: \varnothing\ ,\ \epsilon) \longrightarrow^\star (3 :: \varnothing\ ,\ \epsilon)}{(1 :: 2 :: \varnothing\ ,\ \texttt{Add}) \longrightarrow^\star (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add}) \longrightarrow^\star (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow^\star (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow^\star (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}$$

Question. Evaluate `Push 1 Push 2 Swap Add` in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 2: Combine using the transitivity rule.

Example: Evaluating a Program

$$
\dfrac{(1) \quad \dfrac{(2) \quad \dfrac{(3) \quad \dfrac{(4) \quad \dfrac{}{(3 :: \varnothing\ ,\ \epsilon) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{refl}}{(1 :: 2 :: \varnothing\ ,\ \texttt{Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(2 :: 1 :: \varnothing\ ,\ \texttt{Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(1 :: \varnothing,\ \texttt{Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}}{(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow^{\star} (3 :: \varnothing\ ,\ \epsilon)}\,\texttt{trans}
$$

Question. Evaluate `Push 1 Push 2 Swap Add` in an empty stack. In other words, find a configuration C which cannot be reduced such that

$$(\varnothing,\ \texttt{Push 1 Push 2 Swap Add}) \longrightarrow C$$

Step 2: Combine using the transitivity rule.

Understanding Check

```
push 1 push 2 add push 2 add add
```

Evaluate the above program in the empty stack. Write down the necessary single-step reductions and which rules you used in each case.

A Note on the Project

$$\frac{}{(S,\ \texttt{Push n}\ P) \longrightarrow (n :: S,\ P)}\ \texttt{push}$$

```
| ...
| (s, push n :: rest_prog) -> eval (Int n :: s) rest_prog
| ...
```

Much of what the projects will be is reading these rules and turning them into OCaml code. **It's not always going to be a perfect translation.**

Please take the time to learn how to read and apply these rules.