

CS 314: Principles of Programming Languages

Operational Semantics

Formal Semantics of a Prog. Lang.

- ▶ Mathematical description of the meaning of programs written in that language
 - What a program computes, and what it does
- ▶ **Operational semantics**: define how programs execute
 - Often on an **abstract machine** (mathematical model of computer)
 - Analogous to **interpretation**

This Course: Operational Semantics

- ▶ We will show how an operational semantics may be defined for Micro-Ocaml
 - And develop an interpreter for it, along the way
- ▶ Approach: use **rules** to define a **judgment**

$$e \Rightarrow v$$

- Says “*e evaluates to v*”
- **e**: expression in Micro-OCaml
- **v**: value that results from evaluating **e**

Definitional Interpreter

- ▶ It turns out that the rules for judgment $e \Rightarrow v$ can be easily turned into idiomatic OCaml code
 - The language's expressions e and values v have corresponding OCaml datatype representations `exp` and `value`
 - The semantics is represented as a function

`eval: exp -> value`

- ▶ This way of presenting the semantics is referred to as a **definitional interpreter**
 - The interpreter defines the language's meaning

Micro-OCaml Expression Grammar

$$e ::= x \mid n \mid e + e \mid \text{let } x = e \text{ in } e$$

► e , x , n are *meta-variables* that stand for categories of syntax

- x is any identifier (like z , y , foo)
- n is any numeral (like 1 , 0 , 10 , -25)
- e is any expression (here defined, recursively!)

► *Concrete syntax* of actual expressions in **black**

- Such as let , $+$, z , foo , in , ...

• $::=$ and $|$ are *meta-syntax* used to define the syntax of a language (part of “Backus-Naur form,” or BNF)

Micro-OCaml Expression Grammar

$$e ::= x \mid n \mid e + e \mid \text{let } x = e \text{ in } e$$

Examples

- **1** is a numeral n which is an expression e
- **1+z** is an expression e because
 - **1** is an expression e ,
 - **z** is an identifier x , which is an expression e , and
 - **e + e** is an expression e
- **let z = 1 in 1+z** is an expression e because
 - **z** is an identifier x ,
 - **1** is an expression e ,
 - **1+z** is an expression e , and
 - **let x = e in e** is an expression e

Abstract Syntax = Structure

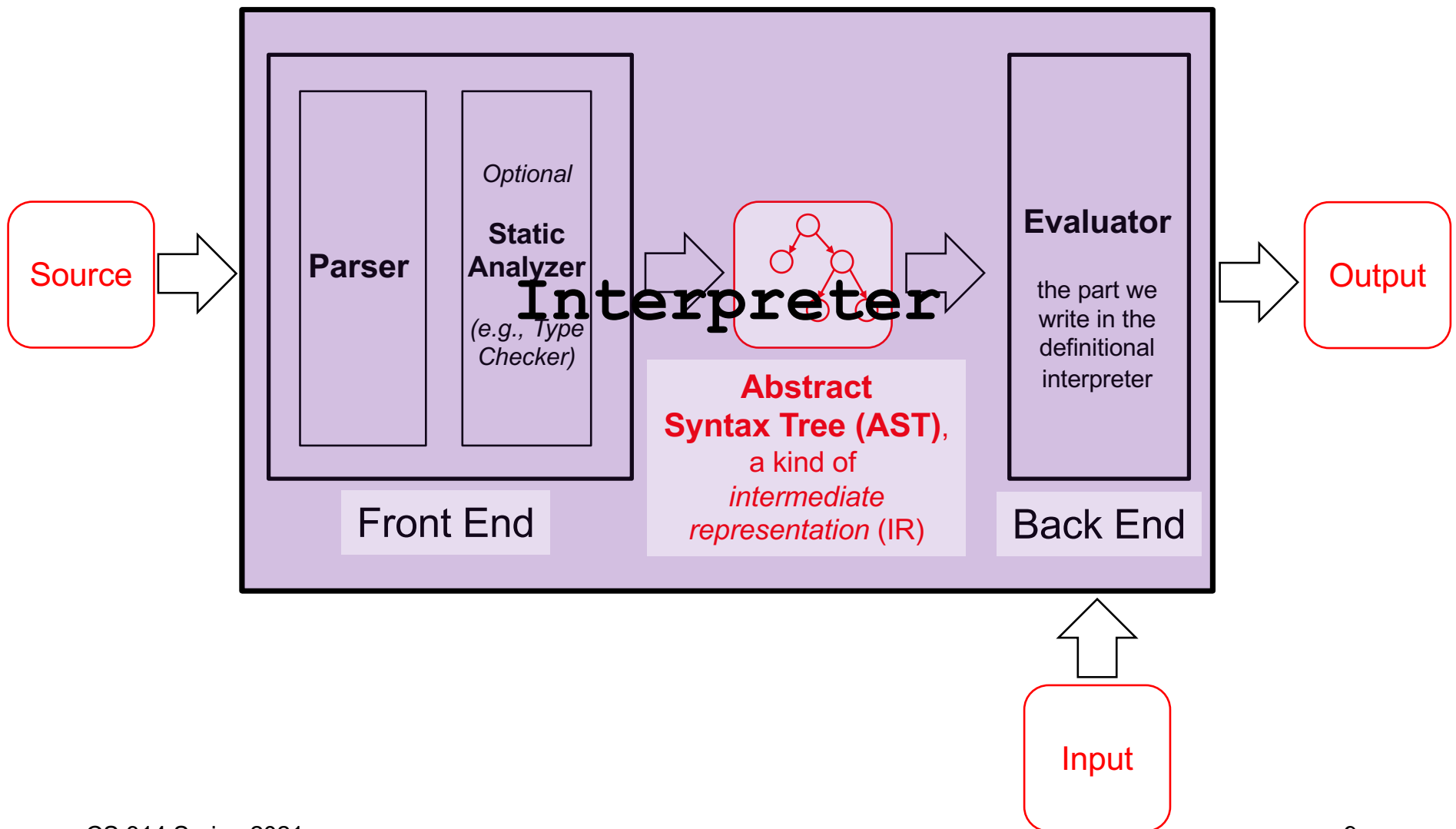
- ▶ Here, the grammar for e is describing its **abstract syntax tree (AST)**, i.e., e 's structure

$e ::= x \mid n \mid e + e \mid \text{let } x = e \text{ in } e$

corresponds to (in definitional interpreter)

```
type id = string
type num = int
type exp =
  | Ident of id           (* x *)
  | Num of num           (* n *)
  | Plus of exp * exp    (* e+e *)
  | Let of id * exp * exp
                        (* let x=e in e *)
```

Aside: Real Interpreters



Values

- ▶ An expression's final result is a **value**. What can values be?

$v ::= n$

- ▶ Just numerals for now
 - In terms of an interpreter's representation:
`type value = int`
 - In a full language, values v will also include booleans (`true`, `false`), strings, functions, ...

Defining the Semantics

- ▶ Use **rules** to define **judgment** $e \Rightarrow v$
- ▶ Judgments are just statements. We use rules to prove that the statement is true.
 - $1+3 \Rightarrow 4$
 - $1+3$ is an expression e , and 4 is a value v
 - This judgment claims that $1+3$ evaluates to 4
 - We use rules to prove it to be true
 - **let foo=1+2 in foo+5 \Rightarrow 8**
 - **let f=1+2 in let z=1 in f+z \Rightarrow 4**

Rules as English Text

No rule when e is x

- ▶ Suppose e is a numeral n
 - Then e evaluates to itself, i.e., $n \Rightarrow n$
- ▶ Suppose e is an addition expression $e1 + e2$
 - If $e1$ evaluates to $n1$, i.e., $e1 \Rightarrow n1$
 - If $e2$ evaluates to $n2$, i.e., $e2 \Rightarrow n2$
 - Then e evaluates to $n3$, where $n3$ is the sum of $n1$ and $n2$
 - i.e., $e1 + e2 \Rightarrow n3$
- ▶ Suppose e is a let expression **let $x = e1$ in $e2$**
 - If $e1$ evaluates to v , i.e., $e1 \Rightarrow v1$
 - If $e2\{v1/x\}$ evaluates to $v2$, i.e., $e2\{v1/x\} \Rightarrow v2$
 - Here, $e2\{v1/x\}$ means “the expression after substituting occurrences of x in $e2$ with $v1$ ”
 - Then e evaluates to $v2$, i.e., **let $x = e1$ in $e2 \Rightarrow v2$**

Rules of Inference

- ▶ We can use a more compact notation for the rules we just presented: **rules of inference**

- Has the following format

$$\frac{H_1 \quad \dots \quad H_n}{C}$$

- Says: if the conditions $H_1 \quad \dots \quad H_n$ (“hypotheses”) are true, then the condition C (“conclusion”) is true
 - If $n=0$ (no hypotheses) then the conclusion automatically holds; this is called an axiom
- ▶ We are using inference rules where C is our judgment about evaluation, i.e., that $e \Rightarrow v$

Rules of Inference: Num and Sum

- ▶ Suppose e is a numeral n
 - Then e evaluates to itself, i.e., $n \Rightarrow n$

$$n \Rightarrow n$$

- ▶ Suppose e is an addition expression $e1 + e2$
 - If $e1$ evaluates to $n1$, i.e., $e1 \Rightarrow n1$
 - If $e2$ evaluates to $n2$, i.e., $e2 \Rightarrow n2$
 - Then e evaluates to $n3$, where $n3$ is the sum of $n1$ and $n2$
 - I.e., $e1 + e2 \Rightarrow n3$

$$\frac{e1 \Rightarrow n1 \quad e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2}{e1 + e2 \Rightarrow n3}$$

Rules of Inference: Let

- ▶ Suppose e is a let expression $\text{let } x = e1 \text{ in } e2$
 - If $e1$ evaluates to v , i.e., $e1 \Rightarrow v1$
 - If $e2\{v1/x\}$ evaluates to $v2$, i.e., $e2\{v1/x\} \Rightarrow v2$
 - Then e evaluates to $v2$, i.e., $\text{let } x = e1 \text{ in } e2 \Rightarrow v2$

$$\frac{e1 \Rightarrow v1 \quad e2\{v1/x\} \Rightarrow v2}{\text{let } x = e1 \text{ in } e2 \Rightarrow v2}$$

Derivations

- ▶ When we apply rules to an expression in succession, we produce a **derivation**
 - It's a kind of **tree**, rooted at the conclusion
- ▶ Produce a derivation by **goal-directed search**
 - Pick a rule that could prove the goal
 - Then repeatedly apply rules on the corresponding hypotheses
 - **Goal: Show that $\text{let } x = 4 \text{ in } x+3 \Rightarrow 7$**

Derivations

$$n \Rightarrow n$$

$$e1 \Rightarrow n1 \quad e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2$$

$$e1 + e2 \Rightarrow n3$$

$$e1 \Rightarrow v1 \quad e2\{v1/x\} \Rightarrow v2$$

$$\text{let } x = e1 \text{ in } e2 \Rightarrow v2$$

Goal: show that

$$\text{let } x = 4 \text{ in } x+3 \Rightarrow 7$$

$$4 \Rightarrow 4 \quad 3 \Rightarrow 3 \quad 7 \text{ is } 4+3$$

$$4 \Rightarrow 4$$

$$4+3 \Rightarrow 7$$

$$\text{let } x = 4 \text{ in } x+3 \Rightarrow 7$$

Quiz 1

What is derivation of the following judgment?

$$2 + (3 + 8) \Rightarrow 13$$

(a)

$$\begin{array}{l} 2 \Rightarrow 2 \quad 3 + 8 \Rightarrow 11 \\ \hline 2 + (3 + 8) \Rightarrow 13 \end{array}$$

(b)

$$\begin{array}{l} 3 \Rightarrow 3 \quad 8 \Rightarrow 8 \\ \hline 3 + 8 \Rightarrow 11 \quad 2 \Rightarrow 2 \\ \hline 2 + (3 + 8) \Rightarrow 13 \end{array}$$

(c)

$$\begin{array}{l} 8 \Rightarrow 8 \\ 3 \Rightarrow 3 \\ 11 \text{ is } 3+8 \\ \hline 2 \Rightarrow 2 \quad 3 + 8 \Rightarrow 11 \quad 13 \text{ is } 2+11 \\ \hline 2 + (3 + 8) \Rightarrow 13 \end{array}$$

Quiz 1

What is derivation of the following judgment?

$$2 + (3 + 8) \Rightarrow 13$$

(a)

$$\begin{array}{l} 2 \Rightarrow 2 \quad 3 + 8 \Rightarrow 11 \\ \hline 2 + (3 + 8) \Rightarrow 13 \end{array}$$

(b)

$$\begin{array}{l} 3 \Rightarrow 3 \quad 8 \Rightarrow 8 \\ \hline 3 + 8 \Rightarrow 11 \quad 2 \Rightarrow 2 \\ \hline 2 + (3 + 8) \Rightarrow 13 \end{array}$$

(c)

$$\begin{array}{l} 8 \Rightarrow 8 \\ 3 \Rightarrow 3 \\ 11 \text{ is } 3+8 \\ \hline 2 \Rightarrow 2 \quad 3 + 8 \Rightarrow 11 \quad 13 \text{ is } 2+11 \\ \hline 2 + (3 + 8) \Rightarrow 13 \end{array}$$

Definitional Interpreter

Trace of evaluation of `eval` function corresponds to a derivation by the rules

- ▶ The style of rules lends itself directly to the implementation of an **interpreter as a recursive function**

```
let rec eval (e:exp):value =
  match e with
  | Ident x -> (* no rule *)
    failwith "no value"
  | Num n -> n
  | Plus (e1,e2) ->
    let n1 = eval e1 in
    let n2 = eval e2 in
    let n3 = n1+n2 in
    n3
  | Let (x,e1,e2) ->
    let v1 = eval e1 in
    let e2' = subst v1 x e2 in
    let v2 = eval e2' in v2
```

$n \Rightarrow n$

$e1 \Rightarrow n1 \quad e2 \Rightarrow n2 \quad n3 \text{ is } n1+n2$

$e1 + e2 \Rightarrow n3$

$e1 \Rightarrow v1 \quad e2\{v1/x\} \Rightarrow v2$

let $x = e1$ **in** $e2 \Rightarrow v2$

Derivations = Interpreter Call Trees

$$\frac{\frac{4 \Rightarrow 4 \quad 3 \Rightarrow 3 \quad 7 \text{ is } 4+3}{4+3 \Rightarrow 7}}{4 \Rightarrow 4} \quad \text{let } x = 4 \text{ in } x+3 \Rightarrow 7$$

Has the same shape as the recursive call tree of the interpreter:

$$\frac{\frac{\text{eval Num } 4 \Rightarrow 4 \quad \text{eval Num } 3 \Rightarrow 3 \quad 7 \text{ is } 4+3}{\text{eval (subst 4 "x" Plus (Ident ("x"), Num 3))} \Rightarrow 7}}{\text{eval Let ("x", Num 4, Plus (Ident ("x"), Num 3))} \Rightarrow 7}$$

Semantics Defines Program Meaning

- ▶ $e \Rightarrow v$ holds if and only if a *proof* can be built
 - Proofs are derivations: axioms at the top, then rules whose hypotheses have been proved to the bottom
 - No proof means $e \not\Rightarrow v$
- ▶ Proofs can be constructed bottom-up
 - In a goal-directed fashion
- ▶ Thus, function $\text{eval } e = \{v \mid e \Rightarrow v\}$
 - Determinism of semantics implies at most one element for any e
- ▶ So: Expression e *means* v

Environment-style Semantics

- ▶ The previous semantics uses substitution to handle variables
 - As we evaluate, we replace all occurrences of a variable x with values it is bound to
- ▶ An alternative semantics, closer to a real implementation, is to use an **environment**
 - As we evaluate, we maintain an explicit map from variables to values, and look up variables as we see them

Environments

- ▶ Mathematically, an environment is a partial function from identifiers to values
 - If A is an environment, and x is an identifier, then $A(x)$ can either be ...
 - ... a value (intuition: the variable has been declared)
 - ... or undefined (intuition: variable has not been declared)
- ▶ An environment can also be thought of as a table

- If A is

Id	Val
x	0
y	2

- then $A(x)$ is 0, $A(y)$ is 2, and $A(z)$ is undefined

Notation, Operations on Environments

- ▶ • is the empty environment (undefined for all ids)
- ▶ If A is an environment then $A, x:v$ is one that extends A with a mapping from x to v

- Sometimes just write $x:v$ instead of $\bullet, x:v$ for brevity
- *NB.* if A maps x to some v' , then that mapping is *shadowed* by the mapping $x:v$

- ▶ Lookup $A(x)$ is defined as follows

$$\bullet(x) = \text{undefined}$$

$$(A, y:v)(x) = \begin{cases} v \\ A(x) \\ \text{undefined} \end{cases}$$

$$\text{if } x = y$$

$$\text{if } x \neq y \text{ and } A(x) \text{ defined}$$

$$\text{otherwise}$$

Definitional Interpreter: Environments

```
type env = (id * value) list

let extend env x v = (x,v)::env

let rec lookup env x =
  match env with
  [] -> failwith "undefined"
| (y,v)::env' ->
  if x = y then v
  else lookup env' x
```

An environment is just a list of mappings,
which are just pairs of variable to value
- called an **association list**

Semantics with Environments

- ▶ The environment semantics changes the judgment

$$e \Rightarrow v$$

to be

$$A; e \Rightarrow v$$

where A is an environment

- Idea: A is used to give values to the identifiers in e
 - A can be thought of as containing declarations made up to e
- ▶ Previous rules can be modified by
 - Inserting A everywhere in the judgments
 - Adding a rule to look up variables x in A
 - Modifying the rule for **let** to add x to A

Environment-style Rules

$$\frac{A(\mathbf{x}) = \mathbf{v}}{A; \mathbf{x} \Rightarrow \mathbf{v}}$$

Look up
variable \mathbf{x} in
environment A

$$\frac{}{A; \mathbf{n} \Rightarrow \mathbf{n}}$$

$$\frac{A; \mathbf{e1} \Rightarrow \mathbf{v1} \quad A, \mathbf{x} : \mathbf{v1}; \mathbf{e2} \Rightarrow \mathbf{v2}}{A; \text{let } \mathbf{x} = \mathbf{e1} \text{ in } \mathbf{e2} \Rightarrow \mathbf{v2}}$$

Extend
environment A
with mapping
from \mathbf{x} to $\mathbf{v1}$

$$\frac{A; \mathbf{e1} \Rightarrow \mathbf{n1} \quad A; \mathbf{e2} \Rightarrow \mathbf{n2} \quad \mathbf{n3} \text{ is } \mathbf{n1} + \mathbf{n2}}{A; \mathbf{e1} + \mathbf{e2} \Rightarrow \mathbf{n3}}$$

Definitional Interpreter: Evaluation

```
let rec eval env e =
  match e with
  | Ident x -> lookup env x
  | Num n -> n
  | Plus (e1,e2) ->
    let n1 = eval env e1 in
    let n2 = eval env e2 in
    let n3 = n1+n2 in
    n3
  | Let (x,e1,e2) ->
    let v1 = eval env e1 in
    let env' = extend env x v1 in
    let v2 = eval env' e2 in v2
```

Quiz 2

What is a derivation of the following judgment?

•; let $x=3$ in $x+2 \Rightarrow 5$

(a)

$$\frac{\begin{array}{ccc} x \Rightarrow 3 & 2 \Rightarrow 2 & 5 \text{ is } 3+2 \\ \hline 3 \Rightarrow 3 & x+2 \Rightarrow 5 & \end{array}}{\text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

(c)

$$\frac{\begin{array}{ccc} x:2; x \Rightarrow 3 & x:2; 2 \Rightarrow 2 & 5 \text{ is } 3+2 \\ \hline \end{array}}{\bullet; \text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

(b)

$$\frac{\begin{array}{ccc} x:3; x \Rightarrow 3 & x:3; 2 \Rightarrow 2 & 5 \text{ is } 3+2 \\ \hline \bullet; 3 \Rightarrow 3 & x:3; x+2 \Rightarrow 5 & \end{array}}{\bullet; \text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

Quiz 2

What is a derivation of the following judgment?

•; let $x=3$ in $x+2 \Rightarrow 5$

(a)

$$\frac{\begin{array}{ccc} x \Rightarrow 3 & 2 \Rightarrow 2 & 5 \text{ is } 3+2 \\ \hline 3 \Rightarrow 3 & x+2 \Rightarrow 5 & \end{array}}{\text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

(c)

$$\frac{\begin{array}{ccc} x:2; x \Rightarrow 3 & x:2; 2 \Rightarrow 2 & 5 \text{ is } 3+2 \\ \hline \end{array}}{\bullet; \text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

(b)

$$\frac{\begin{array}{ccc} x:3; x \Rightarrow 3 & x:3; 2 \Rightarrow 2 & 5 \text{ is } 3+2 \\ \hline \bullet; 3 \Rightarrow 3 & x:3; x+2 \Rightarrow 5 & \end{array}}{\bullet; \text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

Adding Conditionals to Micro-OCaml

```
e ::= x | v | e + e | let x = e in e  
      | eq0 e | if e then e else e  
  
v ::= n | true | false
```

- In terms of interpreter definitions:

```
type exp =  
  | Val of value  
  | ... (* as before *)  
  | Eq0 of exp  
  | If of exp * exp * exp  
  
type value =  
  Int of int  
  | Bool of bool
```

Rules for Eq0 and Booleans

$$A; \text{true} \Rightarrow \text{true}$$
$$A; \text{false} \Rightarrow \text{false}$$
$$A; e \Rightarrow 0$$
$$A; \text{eq0 } e \Rightarrow \text{true}$$
$$A; e \Rightarrow v \quad v \neq 0$$
$$A; \text{eq0 } e \Rightarrow \text{false}$$

- ▶ Booleans evaluate to themselves
 - $A; \text{false} \Rightarrow \text{false}$
- ▶ `eq0` tests for 0
 - $A; \text{eq0 } 0 \Rightarrow \text{true}$
 - $A; \text{eq0 } 3+4 \Rightarrow \text{false}$

Rules for Conditionals

$$A; e1 \Rightarrow \text{true} \quad A; e2 \Rightarrow v$$
$$A; \text{if } e1 \text{ then } e2 \text{ else } e3 \Rightarrow v$$
$$A; e1 \Rightarrow \text{false} \quad A; e3 \Rightarrow v$$
$$A; \text{if } e1 \text{ then } e2 \text{ else } e3 \Rightarrow v$$

- ▶ Notice that only one branch is evaluated
 - $A; \text{if } \text{eq0 } 0 \text{ then } 3 \text{ else } 4 \Rightarrow 3$
 - $A; \text{if } \text{eq0 } 1 \text{ then } 3 \text{ else } 4 \Rightarrow 4$

Quiz 3

What is the derivation of the following judgment?

•; if eq0 3-2 then 5 else 10 \Rightarrow 10

(a)
•; 3 \Rightarrow 3 •; 2 \Rightarrow 2 3-2 is 1

•; eq0 3-2 \Rightarrow false •; 10 \Rightarrow 10

•; if eq0 3-2 then 5 else 10 \Rightarrow 10

(b)
3 \Rightarrow 3 2 \Rightarrow 2
3-2 is 1

eq0 3-2 \Rightarrow false 10 \Rightarrow 10

if eq0 3-2 then 5 else 10 \Rightarrow 10

(c)
•; 3 \Rightarrow 3
•; 2 \Rightarrow 2
3-2 is 1

•; 3-2 \Rightarrow 1 1 \neq 0

•; eq0 3-2 \Rightarrow false •; 10 \Rightarrow 10

•; if eq0 3-2 then 5 else 10 \Rightarrow 10

Quiz 3

What is the derivation of the following judgment?

•; if eq0 3-2 then 5 else 10 \Rightarrow 10

(a)

```
•; 3  $\Rightarrow$  3    •; 2  $\Rightarrow$  2    3-2 is 1
-----
•; eq0 3-2  $\Rightarrow$  false          •; 10  $\Rightarrow$  10
-----
•; if eq0 3-2 then 5 else 10  $\Rightarrow$  10
```

(b)

```
3  $\Rightarrow$  3    2  $\Rightarrow$  2
3-2 is 1
-----
eq0 3-2  $\Rightarrow$  false          10  $\Rightarrow$  10
-----
if eq0 3-2 then 5 else 10  $\Rightarrow$  10
```

(c)

```
•; 3  $\Rightarrow$  3
•; 2  $\Rightarrow$  2
3-2 is 1
-----
•; 3-2  $\Rightarrow$  1    1  $\neq$  0
-----
•; eq0 3-2  $\Rightarrow$  false          •; 10  $\Rightarrow$  10
-----
•; if eq0 3-2 then 5 else 10  $\Rightarrow$  10
```

Updating the Interpreter

```
let rec eval env e =
  match e with
  | Ident x -> lookup env x
  | Val v -> v
  | Plus (e1,e2) ->
    let Int n1 = eval env e1 in
    let Int n2 = eval env e2 in
    let n3 = n1+n2 in
    Int n3
  | Let (x,e1,e2) ->
    let v1 = eval env e1 in
    let env' = extend env x v1 in
    let v2 = eval env' e2 in v2
  | Eq0 e1 ->
    let Int n = eval env e1 in
    if n=0 then Bool true else Bool false
  | If (e1,e2,e3) ->
    let Bool b = eval env e1 in
    if b then eval env e2
    else eval env e3
```

Basically both rules for eq0 in this one snippet

Both if rules here

Quick Look: Type Checking

- ▶ Inference rules can also be used to specify a program's **static semantics**
 - I.e., the rules for type checking
- ▶ We won't cover this in depth in this course, but here is a flavor.
- ▶ **Types** $t ::= \text{bool} \mid \text{int}$
- ▶ Judgment $\vdash e : t$ says e has type t
 - We define inference rules for this judgment, just as with the operational semantics

Some Type Checking Rules

- ▶ Boolean constants have type `bool`

$$\frac{}{\vdash \text{true} : \text{bool}}$$
$$\frac{}{\vdash \text{false} : \text{bool}}$$

- ▶ Equality checking has type `bool` too
 - Assuming its target expression has type `int`

$$\frac{}{\vdash e : \text{int}}$$
$$\frac{}{\vdash \text{eq0 } e : \text{bool}}$$

- ▶ Conditionals

$$\frac{}{\vdash e1 : \text{bool} \quad \vdash e2 : t \quad \vdash e3 : t}$$
$$\frac{}{\vdash \text{if } e1 \text{ then } e2 \text{ else } e3 : t}$$

Handling Binding

- ▶ What about the types of variables?
 - Taking inspiration from the environment-style operational semantics, what could you do?
- ▶ Change judgment to be $G \vdash e : t$ which says *e has type t under type environment G*
 - G is a map from variables x to types t
 - Analogous to map A , but maps vars to types, not values
- ▶ What would be the rules for **let**, and variables?

Type Checking with Binding

- ▶ Variable lookup

$$\frac{G(\mathbf{x}) = \mathbf{t}}{G \vdash \mathbf{x} : \mathbf{t}}$$

analogous to

$$\frac{A(\mathbf{x}) = \mathbf{v}}{A; \mathbf{x} \Rightarrow \mathbf{v}}$$

- ▶ Let binding

$$\frac{G \vdash \mathbf{e1} : \mathbf{t1} \quad G, \mathbf{x} : \mathbf{t1} \vdash \mathbf{e2} : \mathbf{t2}}{G \vdash \text{let } \mathbf{x} = \mathbf{e1} \text{ in } \mathbf{e2} : \mathbf{t2}}$$

analogous to

$$\frac{A; \mathbf{e1} \Rightarrow \mathbf{v1} \quad A, \mathbf{x} : \mathbf{v1}; \mathbf{e2} \Rightarrow \mathbf{v2}}{A; \text{let } \mathbf{x} = \mathbf{e1} \text{ in } \mathbf{e2} \Rightarrow \mathbf{v2}}$$

Scaling up

- ▶ Operational semantics (and similarly styled typing rules) can handle full languages
 - With records, recursive variant types, objects, first-class functions, and more
- ▶ Provides a concise notation for explaining what a language does. Clearly shows:
 - Evaluation order
 - Call-by-value vs. call-by-name
 - Static scoping vs. dynamic scoping
 - ... We may look at more of these later