CMSC 330: Organization 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

- Three main approaches to formal semantics
  - Denotational
  - Operational
  - Axiomatic

Styles of Semantics

- Denotational semantics: translate programs into math!
  - Usually: convert programs into functions mapping inputs to outputs
  - Analogous to compilation
- Operational semantics: define how programs execute
  - Often on an abstract machine (mathematical model of computer)
  - Analogous to interpretation
- Axiomatic semantics
  - Describe programs as predicate transformers, i.e. for converting initial assumptions into guaranteed properties after execution
    - Preconditions: assumed properties of initial states
    - Postcondition: guaranteed properties of final states
  - Logical rules describe how to systematically build up these transformers from programs

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* | *n* | *e* **+** *e* | **let** *x* **=** *e* **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*** | ***n*** | ***e*** + ***e*** | `let` ***x*** `=` ***e*** `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

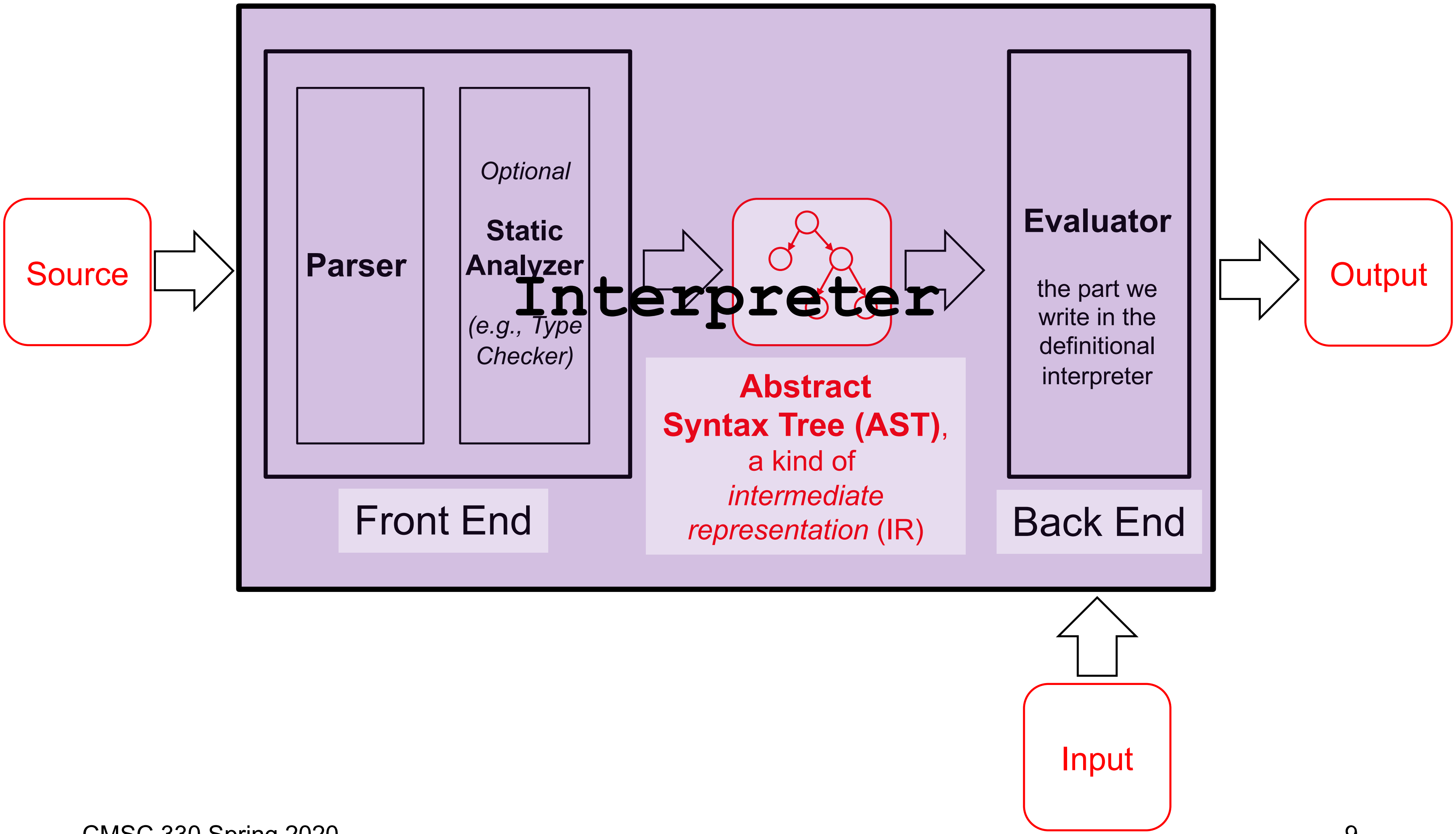
- Here, the grammar for ***e*** is describing its abstract syntax tree (AST), i.e., ***e***'s structure

e ::= ***x*** | ***n*** | ***e*** + ***e*** | **let** ***x*** **=** ***e*** **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*** ⇒ ***v***
- Judgments are just statements. We use rules to prove that the statement is true.
  - **`1+3`** ⇒ **`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`** ⇒ **`8`**
  - **`let f=1+2 in let z=1 in f+z`** ⇒ **`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** ⇒ **n**
- Suppose **e** is an addition expression **e1** + **e2**
  - If **e1** evaluates to **n1**, i.e., **e1** ⇒ **n1**
  - If **e2** evaluates to **n2**, i.e., **e2** ⇒ **n2**
  - Then **e** evaluates to **n3**, where **n3** is the sum of **n1** and **n2**
  - I.e., **e1** + **e2** ⇒ **n3**
- Suppose **e** is a let expression `let x = e1 in e2`
  - If **e1** evaluates to **v**, i.e., **e1** ⇒ **v1**
  - If **e2{v1/x}** evaluates to **v2**, i.e., **e2{v1/x}** ⇒ **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` ⇒ **v2**

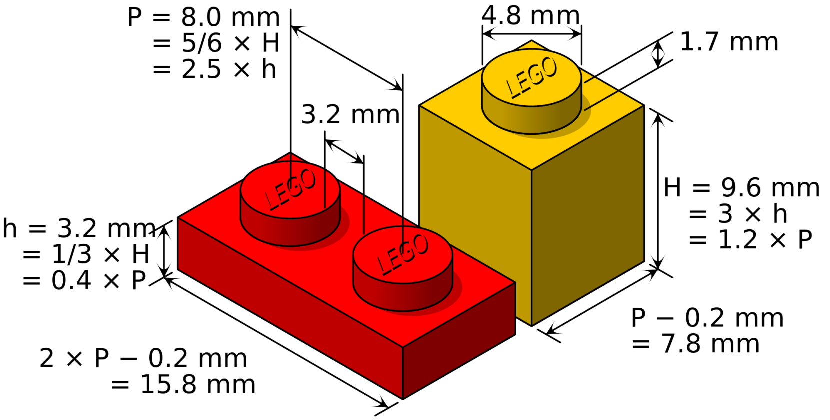
Rules of Inference

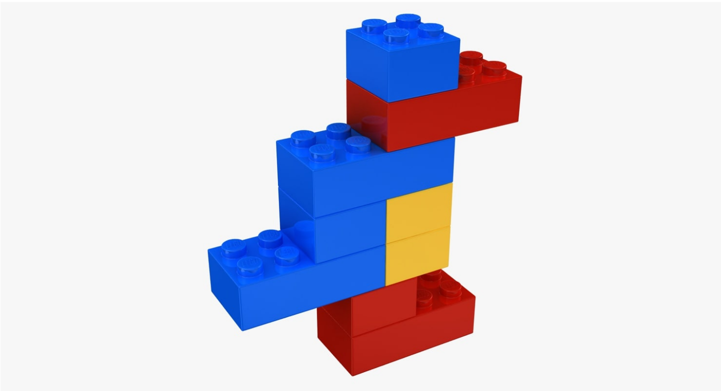
- We can use a more compact notation for the rules we just presented: rules of inference
  - Has the following format

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

  - Says: if the conditions $H_1 \quad \ldots \quad H_n$ ("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 $e \Rightarrow v$

Lego Blocks and Lego Cars





Rules of Inference: Num and Sum

- Suppose ***e*** is a numeral ***n***
  - Then ***e*** evaluates to itself, i.e., ***n*** ⇒ ***n***

$$\frac{}{n \Rightarrow n}$$

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

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

Rules of Inference: Let

- Suppose ***e*** is a let expression **`let`** ***x*** **`=`** ***e1*** **`in`** ***e2***
  - If ***e1*** evaluates to ***v***, i.e., ***e1*** ⇒ ***v1***
  - If ***e2{v1/x}*** evaluates to ***v2***, i.e., ***e2{v1/x}*** ⇒ ***v2***
  - Then ***e*** evaluates to ***v2***, i.e., **`let`** ***x*** **`=`** ***e1*** **`in`** ***e2*** ⇒ ***v2***

$$\frac{e1 \Rightarrow v1 \qquad e2\{v1/x\} \Rightarrow v2}{\texttt{let } x = e1 \texttt{ in } e2 \Rightarrow v2}$$

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 `let x = 4 in x+3` ⇒ `7`

Derivations

$$\frac{}{n \Rightarrow n}$$

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

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

Goal: show that

`let x = 4 in x+3` ⇒ `7`

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

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
```

$$\frac{}{n \Rightarrow n}$$

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

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

Derivations = Interpreter Call Trees

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

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

```
eval Num 4 ⇒ 4    eval Num 3 ⇒ 3    7 is 4+3
-------------------------------------------------
                  eval (subst 4 "x"
eval Num 4 ⇒ 4      Plus(Ident("x"),Num 3)) ⇒ 7
-------------------------------------------------
eval Let("x",Num 4,Plus(Ident("x"),Num 3)) ⇒ 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 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 •,**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

•(**x**) = undefined

$$(A,\ y{:}v)(x) = \begin{cases} v & \text{if } x = y \\ A(x) & \text{if } x \mathrel{<>} y \text{ and } A(x) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$$

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

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

Look up variable x in environment A

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

$$\frac{A;\ e1 \Rightarrow v1 \qquad A,x{:}v1;\ e2 \Rightarrow v2}{A;\ \texttt{let}\ x\ \texttt{=}\ e1\ \texttt{in}\ e2 \Rightarrow v2}$$

Extend environment A with mapping from x to $v1$

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

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
```

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

$$\frac{}{A;\ \texttt{true} \Rightarrow \texttt{true}}$$

$$\frac{A;\ e \Rightarrow 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{true}}$$

$$\frac{}{A;\ \texttt{false} \Rightarrow \texttt{false}}$$

$$\frac{A;\ e \Rightarrow v \qquad v \neq 0}{A;\ \texttt{eq0}\ e \Rightarrow \texttt{false}}$$

- Booleans evaluate to themselves
  - A; **false** ⇒ **false**
- **eq0** tests for 0
  - A; **eq0 0** ⇒ **true**
  - A; **eq0 3+4** ⇒ **false**

Rules for Conditionals

$$\frac{A;\ e1 \Rightarrow \texttt{true} \qquad A;\ e2 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

$$\frac{A;\ e1 \Rightarrow \texttt{false} \qquad A;\ e3 \Rightarrow v}{A;\ \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 \Rightarrow v}$$

- Notice that only one branch is evaluated
  - A; `if eq0 0 then 3 else 4` ⇒ `3`
  - A; `if eq0 1 then 3 else 4` ⇒ `4`

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 ::= **`bool`** | **`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**

$$\frac{}{\vdash \texttt{true} : \texttt{bool}} \qquad \frac{}{\vdash \texttt{false} : \texttt{bool}}$$

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

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

- Conditionals

$$\frac{\vdash e1 : \texttt{bool} \qquad \vdash e2 : t \qquad \vdash e3 : t}{\vdash \texttt{if}\ e1\ \texttt{then}\ e2\ \texttt{else}\ e3 : t}$$

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

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

analogous to

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

- Let binding

$$\frac{G \vdash e1 : t1 \qquad G, x:t1 \vdash e2 : t2}{G \vdash \texttt{let}\ x = e1\ \texttt{in}\ e2 : t2}$$

analogous to

$$\frac{A;\ e1 \Rightarrow v1 \qquad A, x:v1;\ e2 \Rightarrow v2}{A;\ \texttt{let}\ x = e1\ \texttt{in}\ e2 \Rightarrow v2}$$

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

Scaling Up: Lego City