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

```
val 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 \(z, y, \text{foo}\))
  - **\(n\)** is any numeral (like \(1, 0, 10, -25\))
  - **\(e\)** is any expression (here defined, recursively!)

**Concrete syntax** of actual expressions in **black**
- Such as \(\text{let}, +, z, \text{foo}, \text{in}, \ldots\)

- \( ::= \) 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 \( e \)
• \text{let } z = 1 \text{ in } 1+z \text{ is an expression } e \text{ because}
  \- \( z \) is an identifier \( x \),
  \- 1 is an expression \( e \),
  \- \( 1+z \) is an expression \( e \), and
  \- \text{let } x = e \text{ in } e \text{ 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)

```plaintext
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

Source → Parser → Static Analyzer (e.g., Type Checker) → Abstract Syntax Tree (AST), a kind of intermediate representation (IR) → Evaluator → Output

Front End

Back End

Evaluator
the part we write in the definitional interpreter

Input

Interpreter

CMSC 330 Summer 2019
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:
    \[
    \text{type value = int}
    \]
  - 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

- Suppose $e$ is a numeral $n$
  - Then $e$ evaluates to itself, i.e., $n \Rightarrow n$
- Suppose $e$ is an addition expression $e_1 + e_2$
  - If $e_1$ evaluates to $n_1$, i.e., $e_1 \Rightarrow n_1$
  - If $e_2$ evaluates to $n_2$, i.e., $e_2 \Rightarrow n_2$
  - Then $e$ evaluates to $n_3$, where $n_3$ is the sum of $n_1$ and $n_2$
  - I.e., $e_1 + e_2 \Rightarrow n_3$
- Suppose $e$ is a let expression $\text{let } x = e_1 \text{ in } e_2$
  - If $e_1$ evaluates to $v$, i.e., $e_1 \Rightarrow v_1$
  - If $e_2\{v_1/x\}$ evaluates to $v_2$, i.e., $e_2\{v_1/x\} \Rightarrow v_2$
    - Here, $e_2\{v_1/x\}$ means “the expression after substituting occurrences of $x$ in $e_2$ with $v_1$”
  - Then $e$ evaluates to $v_2$, i.e., $\text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2$
Rules of Inference

- We can use a more compact notation for the rules we just presented: rules of inference
  - Has the following format
    \[
    \begin{array}{c}
    H_1 \quad \ldots \quad H_n \\
    \hline
    C
    \end{array}
    \]
  - Says: if the conditions \( H_1 \ldots H_n \) ("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

\[ P = 8.0 \text{ mm} \]
\[ = \frac{5}{6} \times H \]
\[ = 2.5 \times h \]

\[ h = 3.2 \text{ mm} \]
\[ = \frac{1}{3} \times H \]
\[ = 0.4 \times P \]

\[ H = 9.6 \text{ mm} \]
\[ = 3 \times h \]
\[ = 1.2 \times P \]

\[ 2 \times P - 0.2 \text{ mm} \]
\[ = 15.8 \text{ mm} \]
Rules of Inference: Num and Sum

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

- Suppose \( e \) is an addition expression \( e_1 + e_2 \)
  - If \( e_1 \) evaluates to \( n_1 \), i.e., \( e_1 \Rightarrow n_1 \)
  - If \( e_2 \) evaluates to \( n_2 \), i.e., \( e_2 \Rightarrow n_2 \)
  - Then \( e \) evaluates to \( n_3 \), where \( n_3 \) is the sum of \( n_1 \) and \( n_2 \)
  - I.e., \( e_1 + e_2 \Rightarrow n_3 \)

\[
\begin{align*}
  e_1 & \Rightarrow n_1 \\
  e_2 & \Rightarrow n_2 \\
  n_3 & \text{ is } n_1 + n_2 \\
  e_1 + e_2 & \Rightarrow n_3
\end{align*}
\]
Rules of Inference: Let

- Suppose \( e \) is a let expression \( \text{let } x = e_1 \text{ in } e_2 \)
  - If \( e_1 \) evaluates to \( v \), i.e., \( e_1 \Rightarrow v_1 \)
  - If \( e_2\{v_1/x\} \) evaluates to \( v_2 \), i.e., \( e_2\{v_1/x\} \Rightarrow v_2 \)
  - Then \( e \) evaluates to \( v_2 \), i.e., \( \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2 \)

\[
\begin{array}{c|c|c}
 e_1 \Rightarrow v_1 & e_2\{v_1/x\} \Rightarrow v_2 \\
\hline
\text{let } x = e_1 \text{ in } e_2 & \Rightarrow v_2
\end{array}
\]
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 $let \ x = 4 \ in \ x+3 \ \Rightarrow \ 7$
Derivations

\[
\begin{align*}
\text{let } x = 4 \text{ in } x+3 \Rightarrow & \quad 4 \\
\Rightarrow & \quad 4 \\
\Rightarrow & \quad 3 \\
\Rightarrow & \quad 7 \\
\text{is } 4+3
\end{align*}
\]

\[
\begin{align*}
\text{Goal: show that} \\
\text{let } x = 4 \text{ in } x+3 \Rightarrow 7
\end{align*}
\]
Quiz 1

What is derivation of the following judgment?

\[ 2 + (3 + 8) \Rightarrow 13 \]

(a)

\[
\begin{align*}
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(b)

\[
\begin{align*}
3 & \Rightarrow 3 \\
8 & \Rightarrow 8 \\
\hline
3 + 8 & \Rightarrow 11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(c)

\[
\begin{align*}
8 & \Rightarrow 8 \\
3 & \Rightarrow 3 \\
11 & \text{is } 3+8 \\
\hline
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
\hline
13 & \text{is } 2+11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]
Quiz 1

What is derivation of the following judgment?

\[ 2 + (3 + 8) \Rightarrow 13 \]

(a)

\[
\begin{align*}
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(b)

\[
\begin{align*}
3 & \Rightarrow 3 \\
8 & \Rightarrow 8 \\
\hline
3 + 8 & \Rightarrow 11 \\
\hline
2 & \Rightarrow 2 \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]

(c)

\[
\begin{align*}
8 & \Rightarrow 8 \\
3 & \Rightarrow 3 \\
11 & \text{is 3+8} \\
\hline
2 & \Rightarrow 2 \\
3 + 8 & \Rightarrow 11 \\
13 & \text{is 2+11} \\
\hline
2 + (3 + 8) & \Rightarrow 13
\end{align*}
\]
Definitional Interpreter

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

```ml
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
```

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

<table>
<thead>
<tr>
<th><code>e1 ⇒ n1</code></th>
<th><code>e2 ⇒ n2</code></th>
<th><code>n3 is n1+n2</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>e1 + e2 ⇒ n3</code></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><code>e1 ⇒ v1</code></th>
<th><code>e2{v1/x} ⇒ v2</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>let x = e1 in e2 ⇒ v2</code></td>
<td></td>
</tr>
</tbody>
</table>
Derivations = Interpreter Call Trees

\[
\begin{align*}
4 \Rightarrow 4 & \quad 3 \Rightarrow 3 & \quad 7 \text{ is } 4 + 3 \\
4 \Rightarrow 4 & \quad 4 + 3 \Rightarrow 7 \\
\text{let } x = 4 \text{ in } x + 3 \Rightarrow 7
\end{align*}
\]

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

\[
\begin{align*}
\text{eval } \text{Num } 4 \Rightarrow 4 & \quad \text{eval } \text{Num } 3 \Rightarrow 3 & \quad 7 \text{ is } 4 + 3 \\
\text{eval } (\text{subst } 4 \text{ "}x\text{"}) & \quad \text{Plus}(\text{Ident("}x\text{")}, \text{Num } 3)) \Rightarrow 7 \\
\text{eval } \text{Let("}x\text{", Num } 4, \text{Plus}(\text{Ident("}x\text{")}, \text{Num } 3)) \Rightarrow 7
\end{align*}
\]
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 $e \text{ means } v$
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

<table>
<thead>
<tr>
<th>Id</th>
<th>Val</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x$</td>
<td>0</td>
</tr>
<tr>
<td>$y$</td>
<td>2</td>
</tr>
</tbody>
</table>

- 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 $\cdot, x: v$ 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
  $\cdot(x) = \text{undefined}$
  $(A, y: v)(x) = \begin{cases} v & \text{if } x = y \\ A(x) & \text{if } x <> y \text{ and } A(x) \text{ defined} \\ \text{undefined} & \text{otherwise} \end{cases}$
**Definitional Interpreter: Environments**

An environment is just a list of mappings, which are just pairs of variable to value - called an association list.
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 \texttt{let} to add \( x \) to \( A \)
Environment-style Rules

**A(\(x\)) = v**

\[ A; x \Rightarrow v \]

Look up variable \(x\) in environment \(A\)

**A; \(n \Rightarrow n\)**

**A; \(e_1 \Rightarrow v_1\)**

**A; \(e_2 \Rightarrow v_2\)**

**A; let \(x = e_1\) in \(e_2\) \(\Rightarrow v_2\)**

Extend environment \(A\) with mapping from \(x\) to \(v_1\)

**A; \(e_1 \Rightarrow n_1\)**

**A; \(e_2 \Rightarrow n_2\)**

\(n_3\) is \(n_1 + n_2\)

**A; \(e_1 + e_2 \Rightarrow n_3\)**
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 ⇒ 5

(a)  
\[ \begin{align*} 
x & \Rightarrow 3 \\
2 & \Rightarrow 2 \\
5 & \text{is } 3+2 \\
\hline 
3 & \Rightarrow 3 \\
x+2 & \Rightarrow 5 \\
\hline 
\text{let x}=3 \text{ in } x+2 & \Rightarrow 5 
\end{align*} \]

(b)  
\[ \begin{align*} 
x:3; \ x & \Rightarrow 3 \\
x:3; \ 2 & \Rightarrow 2 \\
5 & \text{is } 3+2 \\
\hline 
\text{•; } 3 & \Rightarrow 3 \\
x:3; \ x+2 & \Rightarrow 5 \\
\hline 
\text{•; let x}=3 \text{ in } x+2 & \Rightarrow 5 
\end{align*} \]

(c)  
\[ \begin{align*} 
x:2; \ x & \Rightarrow 3 \\
x:2; \ 2 & \Rightarrow 2 \\
5 & \text{is } 3+2 \\
\hline 
\text{•; let x}=3 \text{ in } x+2 & \Rightarrow 5 
\end{align*} \]
What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

(a)  
\[
\begin{array}{c}
\text{x \Rightarrow 3} \\
\text{2 \Rightarrow 2 \quad 5 \text{ is } 3+2} \\
\hline
\text{3 \Rightarrow 3} \\
\text{x+2 \Rightarrow 5} \\
\end{array}
\] 
-------------------------
\[
\begin{array}{c}
\text{let x=3 in x+2 \Rightarrow 5} \\
\end{array}
\]

(b)  
\[
\begin{array}{c}
\text{x:3; x \Rightarrow 3} \\
\text{x:3; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2} \\
\hline
\text{•; 3 \Rightarrow 3} \\
\text{x:3; x+2 \Rightarrow 5} \\
\end{array}
\] 
---------------------------------------
\[
\begin{array}{c}
\text{•; let x=3 in x+2 \Rightarrow 5} \\
\end{array}
\]

(c)  
\[
\begin{array}{c}
\text{x:2; x\Rightarrow3} \\
\text{x:2; 2\Rightarrow2 \quad 5 \text{ is } 3+2} \\
\hline
\end{array}
\] 
---------------------------------------
\[
\begin{array}{c}
\text{•; let x=3 in x+2 \Rightarrow 5} \\
\end{array}
\]
Adding Conditionals to Micro-OCaml

\[ e ::= \text{\textit{x}} | \text{\textit{v}} | e + e | \text{let } x = e \text{ in } e \]

\[ \text{let } x = e \text{ in } e \]

\[ \text{eq0 } e | \text{if } e \text{ then } e \text{ else } e \]

\[ v ::= n | \text{true} | \text{false} \]

- In terms of interpreter definitions:

```ocaml
type exp =
| Val of value
| ... (* as before *)
| Eq0 of exp
| If of exp * exp * exp
```

```ocaml
type value =
| Int of int
| Bool of bool
```
Rules for Eq0 and Booleans

- 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; \( e_1 \Rightarrow \text{true} \) \quad A; \( e_2 \Rightarrow v \)**
- **\( A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v \)**

- **A; \( e_1 \Rightarrow \text{false} \) \quad A; \( e_3 \Rightarrow v \)**
- **\( A; \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \Rightarrow v \)**

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

What is the derivation of the following judgment?

\[
\text{•; if eq0 3-2 then 5 else 10 } \Rightarrow 10
\]

(a)
\[
\begin{align*}
\text{•; } 3 &\Rightarrow 3 \\
\text{•; } 2 &\Rightarrow 2 \\
3-2 \text{ is 1}
\end{align*}
\]

\[
\begin{align*}
\text{•; } \text{eq0 3-2 } &\Rightarrow \text{false} \\
\text{•; } 10 &\Rightarrow 10
\end{align*}
\]

\[
\begin{align*}
\text{•; if eq0 3-2 then 5 else 10 } &\Rightarrow 10
\end{align*}
\]

(b)
\[
3 \Rightarrow 3 \\
2 \Rightarrow 2 \\
3-2 \text{ is 1}
\]

\[
\begin{align*}
\text{eq0 3-2 } &\Rightarrow \text{false} \\
10 &\Rightarrow 10
\end{align*}
\]

\[
\begin{align*}
\text{if eq0 3-2 then 5 else 10 } &\Rightarrow 10
\end{align*}
\]

(c)
\[
\begin{align*}
\text{•; } 3 &\Rightarrow 3 \\
\text{•; } 2 &\Rightarrow 2 \\
3-2 \text{ is 1}
\end{align*}
\]

\[
\begin{align*}
\text{•; } 3-2 &\Rightarrow 1 \\
1 &\neq 0
\end{align*}
\]

\[
\begin{align*}
\text{•; } \text{eq0 3-2 } &\Rightarrow \text{false} \\
\text{•; } 10 &\Rightarrow 10
\end{align*}
\]

\[
\begin{align*}
\text{•; if eq0 3-2 then 5 else 10 } &\Rightarrow 10
\end{align*}
\]
Quiz 3

What is the derivation of the following judgment?

\[ \text{•}; \text{if eq0 3-2 then 5 else 10} \Rightarrow 10 \]

(a)
\[
\begin{align*}
\text{•}; 3 & \Rightarrow 3 \\
\text{•}; 2 & \Rightarrow 2 \quad 3-2 \text{ is 1} \\
\text{•}; \text{eq0 3-2 } \Rightarrow \text{false} & \quad \text{•}; 10 \Rightarrow 10
\end{align*}
\]

(b)
\[
\begin{align*}
3 & \Rightarrow 3 \\
2 & \Rightarrow 2 \\
3-2 \text{ is 1} & \\
\text{eq0 3-2 } \Rightarrow \text{false} & \quad 10 \Rightarrow 10
\end{align*}
\]

(c)
\[
\begin{align*}
\text{•}; 3 & \Rightarrow 3 \\
\text{•}; 2 & \Rightarrow 2 \\
3-2 \text{ is 1} & \\
\text{•}; \text{eq0 3-2 } \Rightarrow \text{false} & \quad \text{•}; 10 \Rightarrow 10
\end{align*}
\]

\[
\begin{align*}
\text{•}; 3-2 & \Rightarrow 1 \\
1 & \neq 0
\end{align*}
\]

\[
\begin{align*}
\text{•}; \text{if eq0 3-2 then 5 else 10} & \Rightarrow 10
\end{align*}
\]
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
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`
  - `true : bool`
  - `false : bool`

- Equality checking has type `bool` too
  - Assuming its target expression has type `int`
    - `e : int`
    - `eq0 e : bool`

- Conditionals
  - `if e1 then e2 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 $\texttt{let}$, and variables?
Type Checking with Binding

- Variable lookup
  
  \[ G(x) = t \]
  \[ G \vdash x : t \]

- Let binding
  
  \[ G \vdash e_1 : t_1 \]
  \[ G, x : t_1 \vdash e_2 : t_2 \]
  \[ G \vdash \text{let } x = e_1 \text{ in } e_2 : t_2 \]

  \(\text{analogous to}\)

  \[ A(x) = v \]
  \[ A; x \Rightarrow v \]

  \[ A; e_1 \Rightarrow v_1 \]
  \[ A, x : v_1; e_2 \Rightarrow v_2 \]
  \[ A; \text{let } x = e_1 \text{ in } e_2 \Rightarrow v_2 \]
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