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
  - Operational ← **this course**
    - Often on an abstract machine (mathematical model of computer)
    - Analogous to interpretation
  - Denotational
  - Axiomatic

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

- Rules for judgment ***e*** ⇒ ***v*** can be easily turned into idiomatic OCaml code for an interpreter
  - 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

Abstract Syntax Tree spec. via “Grammar”

- We use a grammar for ***e*** to **directly** describe an expression’s 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 *)
```

We are *not* concerned about the process of **parsing**, i.e., from text to an AST. We can thus ignore issues of ambiguity, etc. and focus on the **structure** of the AST given by the grammar

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 (like non-terminals in a CFG)
  - ***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`**, … (like terminals in a CFG)

- ::= 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**

Values

- A value ***v*** is an expression's final result

$$v ::= n$$

- 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

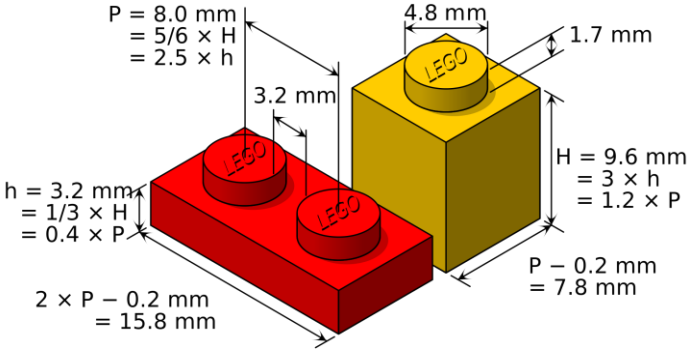
- Use rules to define judgment ***e*** ⇒ ***v***
- Judgments are just statements. We use rules to prove that the statement is true.
  - **`1+3`** ⇒ **`4`**
    - **`1+3`** is an expression ***e***, and **`4`** is a value ***v***
    - This judgment claims that **`1+3`** evaluates to **`4`**
    - We 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`**

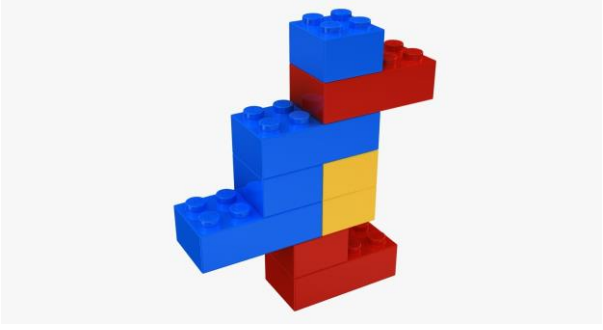
Rules as English Text

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*
  - And 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 *v1*, i.e., *e1* ⇒ *v1*
  - And 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 are Lego Blocks





Rules of Inference

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

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

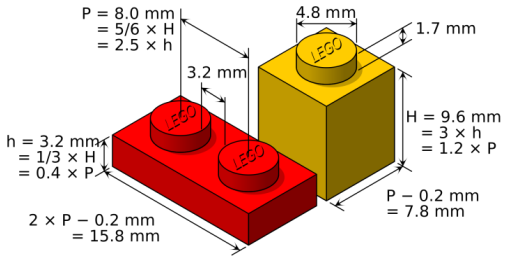
  - Says: if the conditions $H_1 \quad \ldots \quad H_n$ ("hypotheses") are true, then the condition C ("conclusion") is true
  - If n=0 (no hypotheses) then the conclusion automatically holds; this is called an axiom
- We are using inference rules where C is our judgment about evaluation, i.e., that $e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

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***



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

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***

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

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

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

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

Goal: show that

`let x = 4 in x+3` ⇒ `7`

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

Quiz 1

What is derivation of the following judgment?

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

(a)

$$\frac{2 \Rightarrow 2 \qquad 3 + 8 \Rightarrow 11}{2 + (3 + 8) \Rightarrow 13}$$

(b)

$$\frac{\dfrac{8 \Rightarrow 8 \qquad 3 \Rightarrow 3 \qquad 11 \text{ is } 3+8}{3 + 8 \Rightarrow 11} \qquad 2 \Rightarrow 2 \qquad 13 \text{ is } 2+11}{2 + (3 + 8) \Rightarrow 13}$$

(c)

$$\frac{\dfrac{3 \Rightarrow 3 \qquad 8 \Rightarrow 8}{3 + 8 \Rightarrow 11} \qquad 2 \Rightarrow 2}{2 + (3 + 8) \Rightarrow 13}$$

Quiz 1

What is derivation of the following judgment?

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

(a)

$$\frac{2 \Rightarrow 2 \qquad 3 + 8 \Rightarrow 11}{2 + (3 + 8) \Rightarrow 13}$$

(b)

$$\frac{\dfrac{8 \Rightarrow 8 \qquad 3 \Rightarrow 3 \qquad 11 \text{ is } 3+8}{3 + 8 \Rightarrow 11} \qquad 2 \Rightarrow 2 \qquad 13 \text{ is } 2+11}{2 + (3 + 8) \Rightarrow 13}$$

(c)

$$\frac{\dfrac{3 \Rightarrow 3 \qquad 8 \Rightarrow 8}{3 + 8 \Rightarrow 11} \qquad 2 \Rightarrow 2}{2 + (3 + 8) \Rightarrow 13}$$

Definitional Interpreter

- 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
```

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

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

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

Derivations = Interpreter Call Trees

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

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

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

Semantics Defines Program Meaning

- **`e`** ⇒ **`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 there *exists no* **`v`** *for which* **`e`** ⇒ **`v`**
- Proofs can be constructed bottom-up
  - In a goal-directed fashion
- Thus, function eval **`e`** = {**`v`** | **`e`** ⇒ **`v`** }
  - Determinism of semantics implies at most one element for any **`e`**
- So: Expression **`e`** *means* **`v`**

Environment-style Semantics

- So far, semantics used 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 ***v*** (intuition: the value of the variable stored on the stack)
    - undefined (intuition: the variable has not been declared)
- An environment can visualized 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
- A,***x***:***v*** is the environment that extends A with a mapping from ***x*** to ***v***
  - We write ***x***:***v*** instead of •,***x***:***v*** for brevity
  - *NB.* if A maps ***x*** to some ***v′***, then that mapping is *shadowed* by in A,***x***:***v***
- Lookup A(***x***) is defined as follows

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

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

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

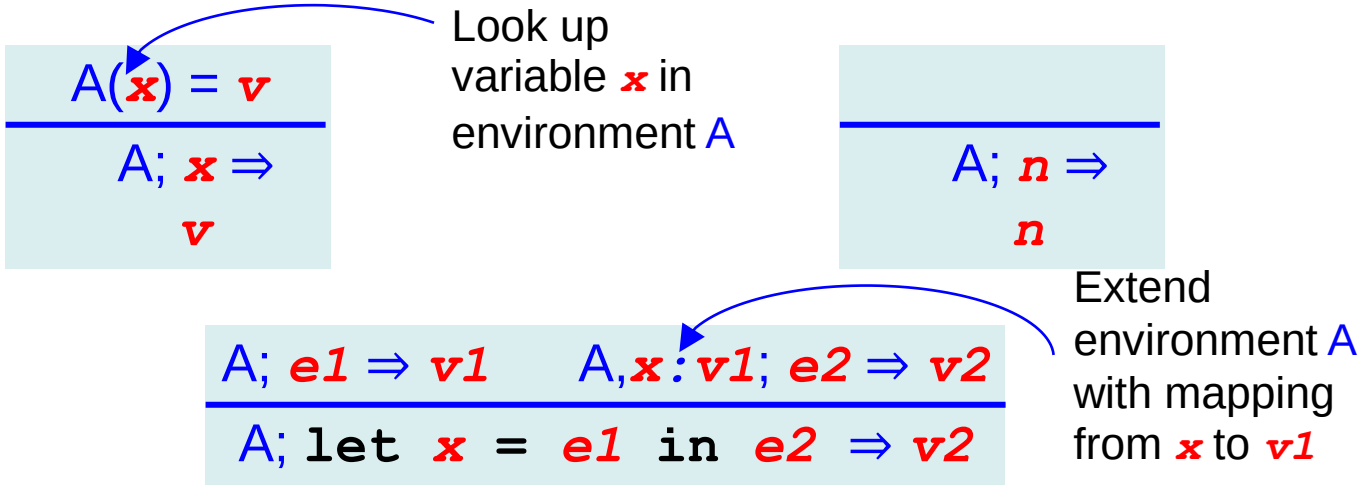
  $$e \Rightarrow v$$

  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



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

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
```

Quiz 2

What is a derivation of the following judgment?

$$\bullet; \texttt{let x=3 in x+2} \Rightarrow 5$$

(a)

$$\frac{x \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\dfrac{3 \Rightarrow 3 \quad x+2 \Rightarrow 5}{\texttt{let x=3 in x+2} \Rightarrow 5}}$$

(b)

$$\frac{\dfrac{x:3; x \Rightarrow 3 \quad x:3; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\bullet; 3 \Rightarrow 3 \quad x:3; x+2 \Rightarrow 5}}{\bullet; \texttt{let x=3 in x+2} \Rightarrow 5}$$

(c)

$$\frac{\dfrac{x:2; x \Rightarrow 3 \quad x:2; 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\bullet; 3 \Rightarrow 3 \quad x:2; x+2 \Rightarrow 5}}{\bullet; \texttt{let x=3 in x+2} \Rightarrow 5}$$

Quiz 2

What is a derivation of the following judgment?

•; let x=3 in x+2 ⇒ 5

(a)

$$\frac{x \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{3 \Rightarrow 3 \quad x+2 \Rightarrow 5} \qquad \frac{3 \Rightarrow 3 \quad x+2 \Rightarrow 5}{\text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

(c)

$$\frac{x:2;\ x \Rightarrow 3 \quad x:2;\ 2 \Rightarrow 2 \quad 5 \text{ is } 3+2 \quad --}{\bullet;\ \text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

(b)

$$\frac{x:3;\ x \Rightarrow 3 \quad x:3;\ 2 \Rightarrow 2 \quad 5 \text{ is } 3+2}{\bullet;3 \Rightarrow 3 \quad x:3;\ x+2 \Rightarrow 5} \qquad \frac{\bullet;3 \Rightarrow 3 \quad x:3;\ x+2 \Rightarrow 5}{\bullet;\ \text{let } x=3 \text{ in } x+2 \Rightarrow 5}$$

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

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

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

- 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`

Quiz 3

What is the derivation of the following judgment?

$$\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10$$

(a)

$$\frac{\dfrac{\bullet;\ 3 \Rightarrow 3 \quad \bullet;\ 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(b)

$$\frac{\dfrac{3 \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad 10 \Rightarrow 10}{\texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(c)

$$\frac{\dfrac{\dfrac{\bullet;\ 3 \Rightarrow 3 \quad \bullet;\ 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\bullet;\ 3\text{-}2 \Rightarrow 1} \qquad 1 \neq 0}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

Quiz 3

What is the derivation of the following judgment?

$$\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10$$

(a)

$$\frac{\dfrac{\bullet;\ 3 \Rightarrow 3 \quad \bullet;\ 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(b)

$$\frac{\dfrac{3 \Rightarrow 3 \quad 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad 10 \Rightarrow 10}{\texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

(c)

$$\frac{\dfrac{\dfrac{\bullet;\ 3 \Rightarrow 3 \quad \bullet;\ 2 \Rightarrow 2 \quad 3\text{-}2 \text{ is } 1}{\bullet;\ \texttt{3-2} \Rightarrow 1} \qquad 1 \neq 0}{\bullet;\ \texttt{eq0 3-2} \Rightarrow \texttt{false}} \qquad \bullet;\ 10 \Rightarrow 10}{\bullet;\ \texttt{if eq0 3-2 then 5 else 10} \Rightarrow 10}$$

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
```

Pattern match will fail if **e1** or **e2** is not an **Int**; this is dynamic type checking! (But **Match_failure** not the best way to signal an error)

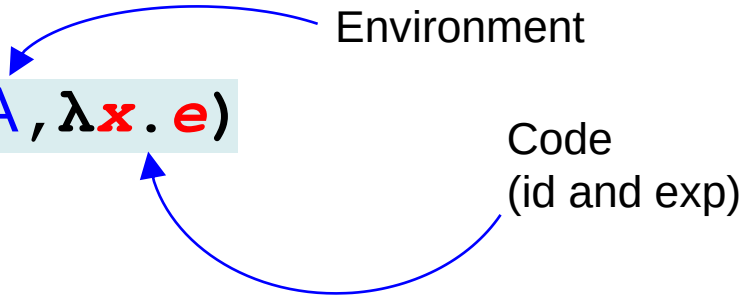
Basically both rules for **eq0** in this one snippet

Both **if** rules here

Adding Closures to Micro-OCaml

```
e ::= x | v | e + e | let x = e in e
    | eq0 e | if e then e else e
    | e e | fun x -> e

v ::= n | true | false | (A,λx.e)
```



- In terms of interpreter definitions:

```
type exp =
  | Val of value
  | If of exp * exp * exp
    … (* as before *)
  | Call of exp * exp
  | Fun of id * exp
```

```
type value =
    Int of int
  | Bool of bool
  | Closure of env * id * exp
```

Rule for Closures: Lexical/Static Scoping

$$\frac{}{A;\ \texttt{fun}\ x\ \texttt{->}\ e\ \Rightarrow\ (A, \lambda x.e)}$$

$$\frac{A;\ e1 \Rightarrow (A', \lambda x.e) \qquad A;\ e2 \Rightarrow v1 \qquad A', x{:}v1;\ e\ \Rightarrow\ v}{A;\ e1\ \ e2\ \Rightarrow\ v}$$

- Notice
  - Creating a closure captures the current environment A
  - A call to a function
    - evaluates the body of the closure's code ***e*** with function closure's environment A' extended with parameter ***x*** bound to argument ***v1***
- Left to you: How will the definitional interpreter change?

Rule for Closures: Dynamic Scoping

$$\frac{}{A;\ \texttt{fun}\ x\ \texttt{->}\ e\ \Rightarrow\ (\bullet, \lambda x.e)}$$

$$\frac{A;\ e1 \Rightarrow (\bullet, \lambda x.e) \qquad A;\ e2 \Rightarrow v1 \qquad A, x{:}v1;\ e \Rightarrow v}{A;\ e1\ \ e2 \Rightarrow v}$$

- Notice
  - Creating a closure ignores the current environment A
  - A call to a function
    - evaluates the body of the closure's code ***e*** with the current environment A extended with parameter ***x*** bound to argument ***v1***
- Easy to see dynamic scoping was an implementation error!

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 ⊢ ***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

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

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 ⊢ ***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

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

analogous to

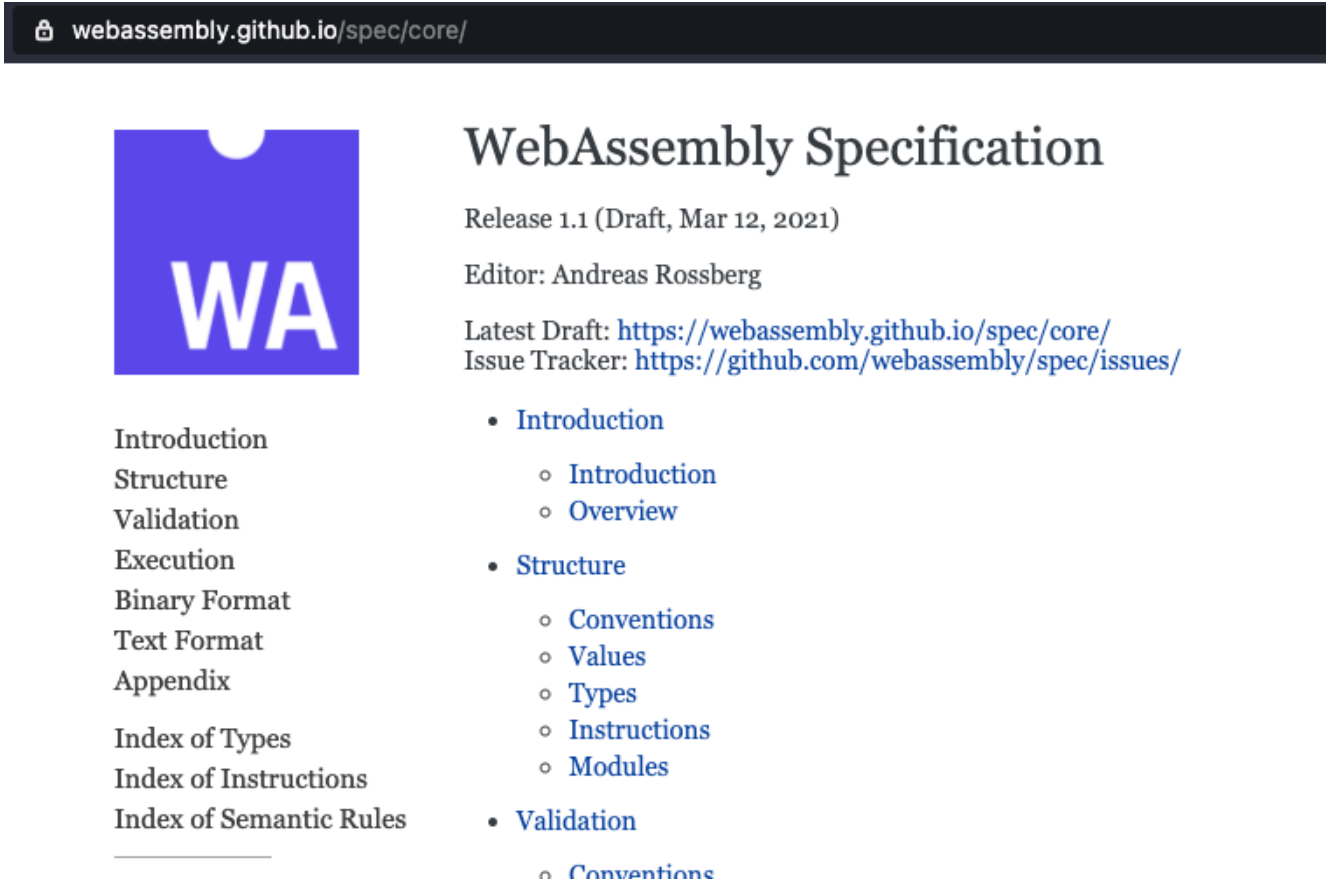
$$\frac{A;\ e1 \Rightarrow v1 \quad A, x:v1;\ e2 \Rightarrow v2}{A;\ \texttt{let}\ x = e1\ \texttt{in}\ e2 \Rightarrow v2}$$

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

Scaling up: Web Assembly



Scaling up: Web Assembly

webassembly.github.io/spec/core/exec/conventions.html#formal-notation

Formal Notation

Note:

This section gives a brief explanation of the notation for specifying execution formally. For the interested reader, a more thorough introduction can be found in respective text books. [2]

The formal execution rules use a standard approach for specifying operational semantics, rendering them into *reduction rules*. Every rule has the following general form:

$$configuration \hookrightarrow configuration$$

A *configuration* is a syntactic description of a program state. Each rule specifies one *step* of execution. As long as there is at most one reduction rule applicable to a given configuration, reduction – and thereby execution – is *deterministic*. WebAssembly has only very few exceptions to this, which are noted explicitly in this specification.

For WebAssembly, a configuration typically is a tuple $(S; F; instr^*)$ consisting of the current store S, the call frame F of the current function, and the sequence of instructions that is to be executed. (A more precise definition is given later.)

To avoid unnecessary clutter, the store S and the frame F are omitted from reduction rules that do not touch them.