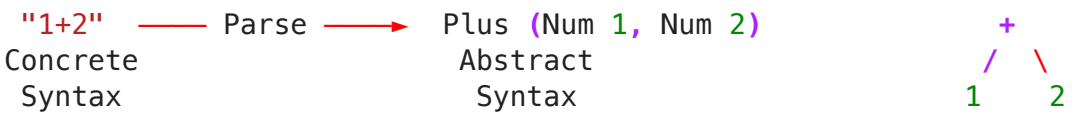
CMSC 330: Organization of Programming Languages

Operational Semantics

Previously in CMSC330

- A **scanner** turns program text into tokens
- A **parser** turns tokens into an **abstract syntax tree** (AST)



Today: **what does `Plus (Num 1, Num 2)` mean?**

In class lecture notebook: https://gitlab.cs.umd.edu/cmsc330-mamat/fall2026/templates/lecture/lectures

Formal Semantics of a Programming Language

A **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* (a mathematical model of a computer)
  - Analogous to interpretation
- We will define an operational semantics for **Micro-OCaml**
  - And develop an interpreter for it, along the way
- Approach: use **rules** to define a **judgment**

$$e \Rightarrow v$$

read "expression e evaluates to value v".

Judgment

A **judgment** is a claim about how a program behaves — a claim we *prove* using rules.

Example

Consider the expression $3 + 4$. To claim that it evaluates to 7, we write the judgment

$$3 + 4 \Rightarrow 7$$

read "$3 + 4$ evaluates to 7".

It is a judgment because it asserts a fact about evaluation; the rules of the semantics are what let us prove it.

Defining the Semantics

We define the judgment $e \Rightarrow v$ using **inference rules**. An inference rule tells us how to prove a judgment from other judgments:

$$\frac{H_1 \quad H_2 \quad \cdots \quad H_n}{C} \qquad\qquad \frac{\begin{array}{c}\text{man}(x) \to \text{mortal}(x) \\ \text{man}(\text{Socrates})\end{array}}{\text{mortal}(\text{Socrates})}$$

The line separates two parts:

- **Premises** $H_1, \ldots, H_n$ (above the line): facts that must be established first.
- **Conclusion** C (below the line): the fact we can establish using those premises.

A rule means $H_1 \wedge H_2 \wedge \cdots \wedge H_n \implies C$: "if all the premises hold, then the conclusion holds." In the example on the right, since Socrates is a man, and all men are mortal, we conclude that Socrates is mortal.

A rule with no premises is an **axiom**: its conclusion always holds.

Rules are Lego Blocks

- Judgments are the LEGO blocks — individual facts or statements.
- Inference rules are the building instructions — they tell you which blocks you need and how to combine them.

- A derivation is the completed LEGO structure — the proof you build by applying the rules.



Micro-OCaml Expression Grammar

`e ::= x | n | e + e | let x = e1 in e2`
Corresponding AST:

In [ ]:
```
type id = string;;

type exp =
  | Ident of id               (* x *)
  | Num of int                (* n *)
  | Plus of exp * exp         (* e+e *)
  | Let of id * exp * exp;;   (* let x=e1 in e2 *)
```

We skip the parser today and write ASTs directly. For example, `let x = 4 in x+3` is:

In [ ]:
```
let ex1 = Let ("x", Num 4, Plus (Ident "x", Num 3));;
```

To print expressions back in concrete syntax, a small pretty printer (nested `+` and `let` are parenthesized):

In [ ]:
```
let rec show (e : exp) : string =
  match e with
  | Ident x -> x
  | Num n -> string_of_int n
  | Plus (e1, e2) -> paren e1 ^ "+" ^ paren e2
  | Let (x, e1, e2) -> "let " ^ x ^ " = " ^ show e1 ^ " in " ^ show e2

and paren e =
  match e with
  | Plus _ | Let _ -> "(" ^ show e ^ ")"
  | _ -> show e;;

show ex1;;
```

Rules of Inference: Num and Sum

$$\frac{}{n \Rightarrow n}\ (\text{axiom}) \qquad \frac{e_1 \Rightarrow n_1 \qquad e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{e_1 + e_2 \Rightarrow n_3}$$

- A number evaluates to itself

- To evaluate `e1 + e2`: evaluate `e1` to `n1`, evaluate `e2` to `n2`, and add them

Read as code, each rule is one case of a `match`:

```
match e with
| Num n -> n
| Plus (e1,e2) ->
    let n1 = eval e1 in
    let n2 = eval e2 in
    let n3 = n1 + n2 in
    n3
```

Rules of Inference: Let

$$\frac{e_1 \Rightarrow v_1 \qquad e_2\{v_1/x\} \Rightarrow v_2}{\texttt{let } x \texttt{ = } e_1 \texttt{ in } e_2 \Rightarrow v_2}$$

$e_2\{v_1/x\}$ means "e_2 with every occurrence of x replaced by v_1" — **substitution**.

```
match e with
| Let (x,e1,e2) ->
    let v1 = eval e1 in
    let e2' = subst v1 x e2 in
    let v2 = eval e2' in
    v2
```

`subst v x e` walks the AST, replacing `Ident x` by `Num v`. One subtlety: in `let x = ... in e2` the inner `let` **shadows** `x`, so we must not substitute into its body.

In [ ]:
```
let rec subst (v : int) (x : id) (e : exp) : exp =
  match e with
  | Ident y -> if x = y then Num v else e
  | Num _ -> e
  | Plus (e1, e2) -> Plus (subst v x e1, subst v x e2)
  | Let (y, e1, e2) ->
    (* the new binding of y hides x inside e2 when y = x *)
    Let (y, subst v x e1, if x = y then e2 else subst v x e2);;
```

In [ ]:
```
show (subst 4 "x" (Plus (Ident "x", Num 3)));;      (* x+3{4/x} *)
```

In [ ]:
```
(* the inner let rebinds x, so only the outer x+... is replaced *)
show (subst 4 "x" (Plus (Ident "x", Let ("x", Num 1, Ident "x"))));;
```

Derivations

- When we apply rules to an expression in succession, we produce a **derivation**

    - It's a kind of **tree**, rooted at the conclusion (at the bottom)
- 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`

- The goal is a `let`, so use the Let rule: we need `4 ⇒ v1` and `x+3{v1/x} ⇒ 7`
- `4 ⇒ 4` by the Num axiom, so `v1 = 4` and `x+3{4/x}` is `4+3`
- `4+3` is a sum, so use the Sum rule: `4 ⇒ 4`, `3 ⇒ 3`, and `7 is 4+3`

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

What is the derivation of the following judgment?

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

- The goal is a sum, so use the Sum rule: we need `2 ⇒ n1`, `3+8 ⇒ n2`, and `13 is n1+n2`
- `2 ⇒ 2` by the Num axiom, so `n1 = 2`
- `3+8` is a sum, so use the Sum rule again: `3 ⇒ 3`, `8 ⇒ 8`, and `11 is 3+8`, so `n2 = 11`
- Finally, `13 is 2+11`

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

Definitional Interpreter

The style of the rules lends itself directly to the implementation of an **interpreter as a recursive function**: one `match` case per rule.

- `Ident x` has **no rule**: with substitution, a variable that is still around when we evaluate it was never bound, so there is no value for it.

In [ ]:
```
type value = int;;

let rec eval (e : exp) : value =
  match e with
  | Ident x -> (* no rule *)
    failwith ("no value for " ^ x)
```

```
  | Num n -> n                                    (* n ⇒ n *)
  | Plus (e1, e2) ->                              (* e1 + e2 ⇒ n3 *)
    let n1 = eval e1 in
    let n2 = eval e2 in
    let n3 = n1 + n2 in
    n3
  | Let (x, e1, e2) ->                            (* let x = e1 in e2 ⇒ v2 *
    let v1 = eval e1 in
    let e2' = subst v1 x e2 in
    let v2 = eval e2' in
    v2;;
```

In [ ]:
```
eval ex1;;     (* let x = 4 in x+3 *)
```

In [ ]:
```
eval (Plus (Num 2, Plus (Num 3, Num 8)));;     (* Quiz 1: 2 + (3 + 8) *)
```

In [ ]:
```
(* let x = 1 in let x = x + 1 in x + x *)
eval (Let ("x", Num 1, Let ("x", Plus (Ident "x", Num 1), Plus (Ident "x", I
```

In [ ]:
```
eval (Plus (Ident "y", Num 1));;     (* Failure: y is not bound -- no rule a
```

Derivations = Interpreter Call Trees

The derivation

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

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

```
                    eval Num 3 ⇒ 3   eval Num 8 ⇒ 8   11 is 3+8
                    ---------------------------------------
eval Num 2 ⇒ 2      eval (Plus (Num 3, Num 8)) ⇒ 11
13 is 2+11
-------------------------------------------------------------
-------------
eval (Plus (Num 2, Plus (Num 3, Num 8))) ⇒ 13
```

Let's make the interpreter print its own derivation. Each judgment is printed when its `eval` call returns, indented by the depth of that call: a judgment's hypotheses appear just above it, indented one level further.

In [ ]:
```
let depth = ref 0;;

(* run f one level deeper, then print the judgment for its result *)
let rule (judgment : 'a -> string) (f : unit -> 'a) : 'a =
  let d = !depth in
  depth := d + 1;
  let v = Fun.protect ~finally:(fun () -> depth := d) f in
```

```
  print_endline (String.make (4 * d) ' ' ^ judgment v);
  v;;

let rec eval_trace (e : exp) : value =
  rule (fun v -> show e ^ " ⇒ " ^ string_of_int v) (fun () ->
    match e with
    | Ident x -> failwith ("no value for " ^ x)
    | Num n -> n
    | Plus (e1, e2) ->
      let n1 = eval_trace e1 in
      let n2 = eval_trace e2 in
      n1 + n2
    | Let (x, e1, e2) ->
      let v1 = eval_trace e1 in
      eval_trace (subst v1 x e2));;
```

In [ ]:
```
eval_trace ex1;;
```

In [ ]:
```
eval_trace (Plus (Num 2, Plus (Num 3, Num 8)));;     (* compare with Quiz 1
```

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, down to the bottom
  - No proof means there *exists no* v *for which* $e \Rightarrow v$ (e.g. `y+1` above)
- Proofs can be constructed bottom-up
  - In a goal-directed fashion
- Thus, function $eval\, e = \{ v \mid e \Rightarrow v \}$
  - Determinism of the semantics implies at most one element for any e
- So: **expression** e **means** v

Environment-style Semantics

- So far, the semantics used **substitution** to handle variables
  - As we evaluate, we replace all occurrences of a variable `x` with the value 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 $A(x)$ is either
  - a value v (intuition: the value of the variable stored on the stack), or
  - undefined (intuition: the variable has not been declared)
- An environment can be 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

- $\bullet$ is the **empty** environment
- $A, x{:}v$ is the environment that **extends** A with a mapping from x to v
  - Sometimes just write $x{:}v$ instead of $\bullet, x{:}v$ for brevity
- **Lookup** $A(x)$ is defined as follows

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

The **most recent** mapping for a variable wins — this is how shadowing works.

Definitional Interpreter: Environments

An environment is just a list of mappings, which are just pairs of variable to value — called an **association list**. The newest mapping is at the head of the list.

In [ ]:
```
type env = (id * value) list;;

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

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

In [ ]:
```
let a : env = extend (extend [] "x" 0) "y" 2;;     (* x:0, y:2 *)
lookup a "x";;
lookup a "y";;
```

In [ ]:
```
lookup a "z";;     (* Failure: undefined *)
```

In [ ]:
```
lookup (extend a "x" 5) "x";;     (* the newer x:5 shadows x:0 *)
```

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

Environment-style Rules

$$\frac{A(x) = v}{A;\ x \Rightarrow v}\ (\text{look up } x \text{ in } A) \qquad \frac{}{A;\ n \Rightarrow n}$$

$$\frac{A;\ e_1 \Rightarrow v_1 \qquad A, x{:}v_1;\ e_2 \Rightarrow v_2}{A;\ \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 \Rightarrow v_2}\ (\text{extend } A \text{ with } x{:}v_1)$$

$$\frac{A;\ e_1 \Rightarrow n_1 \qquad A;\ e_2 \Rightarrow n_2 \qquad n_3 \text{ is } n_1 + n_2}{A;\ e_1 + e_2 \Rightarrow n_3}$$

Variables now have a rule, and `let` no longer rewrites its body.

Definitional Interpreter: Evaluation

In [ ]:
```
let rec eval (env : env) (e : exp) : value =
  match e with
  | Ident x -> lookup env x                          (* A(x) = v *)
  | 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                    (* A, x:v1 *)
    let v2 = eval env' e2 in
    v2;;
```

In [ ]:
```
eval [] ex1;;     (* •; let x = 4 in x+3 *)
```

In [ ]:
```
(* let x = 1 in let x = x + 1 in x + x -- same answer as with substitution *
eval [] (Let ("x", Num 1, Let ("x", Plus (Ident "x", Num 1), Plus (Ident "x"
```

In [ ]:
```
eval a (Plus (Ident "x", Ident "y"));;     (* x:0, y:2; x+y *)
```

Quiz 2

What is a derivation of the following judgment?

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

- The goal is a `let`, so use the Let rule: we need `•; 3 ⇒ v1` and `•, x:v1; x+2 ⇒ 5`
- `•; 3 ⇒ 3` by the Num axiom, so `v1 = 3` and the body is evaluated in `x:3`
- `x+2` is a sum, so use the Sum rule: `x:3; x ⇒ 3`, `x:3; 2 ⇒ 2`, and `5 is 3+2`
- `x:3; x ⇒ 3` by the variable rule, since `(x:3)(x) = 3`

$$\dfrac{\bullet;\ 3 \Rightarrow 3 \qquad \dfrac{\dfrac{(x{:}3)(x) = 3}{x{:}3;\ \mathtt{x} \Rightarrow 3} \qquad x{:}3;\ 2 \Rightarrow 2 \qquad 5 \text{ is } 3+2}{x{:}3;\ \mathtt{x+2} \Rightarrow 5}}{\bullet;\ \mathtt{let\ x=3\ in\ x+2} \Rightarrow 5}$$

In [ ]:
```
(* •, x:3 prints as "x:3"; the newest binding is rightmost, as in the notati
let show_env (env : env) =
  if env = [] then "•"
  else String.concat ", "
      (List.rev_map (fun (x, v) -> x ^ ":" ^ string_of_int v) env);;

let rec eval_trace (env : env) (e : exp) : value =
  rule (fun v -> show_env env ^ "; " ^ show e ^ " ⇒ " ^ string_of_int v) (fu
    match e with
    | Ident x -> lookup env x
    | Num n -> n
    | Plus (e1, e2) ->
      let n1 = eval_trace env e1 in
      let n2 = eval_trace env e2 in
      n1 + n2
    | Let (x, e1, e2) ->
      let v1 = eval_trace env e1 in
      eval_trace (extend env x v1) e2);;
```

In [ ]:
```
eval_trace [] (Let ("x", Num 3, Plus (Ident "x", Num 2)));;
```

Quiz 3

Describe in English what the operator **myst** does:

$$\dfrac{A;\ e_1 \Rightarrow \mathit{true} \qquad A;\ e_2 \Rightarrow \mathit{true}}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow \mathit{false}} \qquad \dfrac{A;\ e_1 \Rightarrow \mathit{false} \qquad A;\ e_2 \Rightarrow \mathit{false}}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow \mathit{false}}$$

$$\frac{A;\ e_1 \Rightarrow true \qquad A;\ e_2 \Rightarrow false}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow true} \qquad \frac{A;\ e_1 \Rightarrow false \qquad A;\ e_2 \Rightarrow true}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow true}$$

Write the four rules as four `match` cases, then look at the truth table:

In [ ]:
```
let myst b1 b2 =
  match b1, b2 with
  | true, true -> false
  | false, false -> false
  | true, false -> true
  | false, true -> true;;

List.map (fun (b1, b2) -> (b1, b2, myst b1 b2))
  [(true, true); (true, false); (false, true); (false, false)];;
```

Answer: **XOR** — true exactly when the two operands differ.

Quiz 4

Describe what the operator **myst** does:

$$\frac{A;\ e_1 \Rightarrow true \qquad A;\ e_2 \Rightarrow true}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow true} \qquad \frac{A;\ e_1 \Rightarrow true \qquad A;\ e_2 \Rightarrow false}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow false}$$

$$\frac{A;\ e_1 \Rightarrow false \qquad A;\ e_2 \Rightarrow true}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow false} \qquad \frac{A;\ e_1 \Rightarrow false \qquad A;\ e_2 \Rightarrow false}{A;\ e_1\ \mathbf{myst}\ e_2 \Rightarrow false}$$

Try it: write these rules as a function and check its truth table.

In [ ]:
```
(* your myst here *)
```

Answer: **AND** — true only when both operands are true.

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:

- Values are no longer just integers, so `value` becomes a variant
- `Num of int` is replaced by `Val of value`, which covers integer and boolean constants

In [30]:
```
type value =
  | Int of int
  | Bool of bool;;

type exp =
  | Ident of id
  | Val of value
  | Plus of exp * exp
  | Let of id * exp * exp
  | Eq0 of exp
  | If of exp * exp * exp;;

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

Out[30]:
```
type value = Int of int | Bool of bool
```

Out[30]:
```
type exp =
    Ident of id
  | Val of value
  | Plus of exp * exp
  | Let of id * exp * exp
  | Eq0 of exp
  | If of exp * exp * exp
```

Out[30]:
```
type env = (id * value) list
```

Rules for Eq0 and Booleans

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

Rules for Conditionals

$$\frac{A;\ e_1 \Rightarrow \texttt{true} \qquad A;\ e_2 \Rightarrow v}{A;\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v} \qquad \frac{A;\ e_1 \Rightarrow \texttt{false} \qquad A;\ e_3 \Rightarrow v}{A;\ \texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Rightarrow v}$$

- Notice that **only one branch is evaluated**

Updating the Interpreter

The slides write `let Int n1 = eval env e1 in ...`. That pattern can fail (what if `e1` evaluates to a boolean?), so OCaml warns that it is not exhaustive. Here we `match` instead and raise a `TypeError` when no rule applies — for example `1 + true`.

In [31]:
```
exception TypeError of string;;

let rec eval (env : env) (e : exp) : value =
  match e with
  | Ident x -> lookup env x
  | Val v -> v
```

```
  | Plus (e1, e2) ->
    (match eval env e1, eval env e2 with
     | Int n1, Int n2 -> Int (n1 + n2)
     | _ -> raise (TypeError "+ expects integers"))
  | 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 ->
    (match eval env e1 with
     | Int n -> if n = 0 then Bool true else Bool false
     | _ -> raise (TypeError "eq0 expects an integer"))
  | If (e1, e2, e3) ->
    (match eval env e1 with
     | Bool b -> if b then eval env e2 else eval env e3     (* only one brar
     | _ -> raise (TypeError "if expects a boolean"));;
```

```
Out[31]: exception TypeError of string
```

```
Out[31]: val eval : env -> exp -> value = <fun>
```

```
In [32]: (* let x = 0 in if eq0 x then 10 else 20 *)
eval [] (Let ("x", Val (Int 0), If (Eq0 (Ident "x"), Val (Int 10), Val (Int
```

```
Out[32]: - : value = Int 10
```

```
In [33]: eval [] (Plus (Val (Int 1), Val (Bool true)));;     (* TypeError: no rule ap
```

```
Exception: TypeError "+ expects integers".

Raised at eval in file "[31]", line 10, characters 12-50
Called from <unknown> in file "[33]", line 1, characters 0-45
Called from Topeval.load_lambda in file "toplevel/byte/topeval.ml", line 93,
characters 4-14
```

```
In [34]: (* if eq0 1 then 1 + true else 10 -- the bad branch is never evaluated *)
eval [] (If (Eq0 (Val (Int 1)), Plus (Val (Int 1), Val (Bool true)), Val (Ir
```

```
Out[34]: - : value = Int 10
```

Quiz 5

What is the derivation of the following judgment? (Subtraction has a rule just like the one for `+`.)

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

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

```
----------------------------------------
•; if eq0 3-2 then 5 else 10 ⇒ 10
```

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)
```

A function value is a **closure** $(A, \lambda x.\, e)$: the **code** (parameter x and body e) together with an **environment** A.

In terms of interpreter definitions: a closure holds an `env`, an `env` holds `value`s, and a closure's code is an `exp` — so the three types are **mutually recursive** and are defined together with `type … and …`.

In [38]:
```
type exp =
  | Ident of id
  | Val of value
  | Plus of exp * exp
  | Let of id * exp * exp
  | Eq0 of exp
  | If of exp * exp * exp
  | Call of exp * exp                  (* e e *)
  | Fun of id * exp                    (* fun x -> e *)

and value =
  | Int of int
  | Bool of bool
  | Closure of env * id * exp          (* (A, λx.e) *)

and env = (id * value) list;;
```

Out[38]:
```
type exp =
    Ident of id
  | Val of value
  | Plus of exp * exp
  | Let of id * exp * exp
  | Eq0 of exp
  | If of exp * exp * exp
  | Call of exp * exp
  | Fun of id * exp
and value = Int of int | Bool of bool | Closure of env * id * exp
and env = (id * value) list
```

Rule for Closures: Lexical/Static Scoping

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

$$\frac{A;\ e_1 \Rightarrow (A', \lambda x.\, e) \qquad A;\ e_2 \Rightarrow v_1 \qquad A', x{:}v_1;\ e \Rightarrow v}{A;\ e_1\ e_2 \Rightarrow v}$$

Notice

- Creating a closure **captures the current environment** A
- A call to a function evaluates the body e of the closure's code with the **closure's environment** A' extended with parameter x bound to argument v_1

In [39]:
```
let rec eval (env : env) (e : exp) : value =
  match e with
  | Ident x -> lookup env x
  | Val v -> v
  | Plus (e1, e2) ->
    (match eval env e1, eval env e2 with
     | Int n1, Int n2 -> Int (n1 + n2)
     | _ -> raise (TypeError "+ expects integers"))
  | Let (x, e1, e2) ->
    let v1 = eval env e1 in
    eval (extend env x v1) e2
  | Eq0 e1 ->
    (match eval env e1 with
     | Int n -> Bool (n = 0)
     | _ -> raise (TypeError "eq0 expects an integer"))
  | If (e1, e2, e3) ->
    (match eval env e1 with
     | Bool b -> if b then eval env e2 else eval env e3
     | _ -> raise (TypeError "if expects a boolean"))
  | Fun (x, body) -> Closure (env, x, body)          (* capture A *)
  | Call (e1, e2) ->
    (match eval env e1 with
     | Closure (env', x, body) ->
       let v1 = eval env e2 in
       eval (extend env' x v1) body                 (* A', x:v1 *)
     | _ -> raise (TypeError "call of a non-function"));;
```

Out[39]:
```
val eval : env -> exp -> value = <fun>
```

In [40]:
```
(* let f = fun x -> x + 1 in f 41 *)
eval [] (Let ("f", Fun ("x", Plus (Ident "x", Val (Int 1))),
             Call (Ident "f", Val (Int 41))));;
```

Out[40]:
```
- : value = Int 42
```

In [41]:
```
(* the value of a fun expression is a closure *)
eval [("y", Int 5)] (Fun ("x", Plus (Ident "x", Ident "y")));;
```

Out[41]:
```
- : value = Closure ([("y", Int 5)], "x", Plus (Ident "x", Ident "y"))
```

Rule for Closures: Dynamic Scoping

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

$$\frac{A;\ e_1 \Rightarrow (\bullet, \lambda x.\, e) \qquad A;\ e_2 \Rightarrow v_1 \qquad A, x{:}v_1;\ e \Rightarrow v}{A;\ e_1\ e_2 \Rightarrow v}$$

Notice

- Creating a closure **ignores** the current environment A
- A call to a function evaluates the body e of the closure's code with the **current environment** A extended with parameter x bound to argument v_1

Only two cases change:

In [42]:
```
let rec eval_dyn (env : env) (e : exp) : value =
  match e with
  | Ident x -> lookup env x
  | Val v -> v
  | Plus (e1, e2) ->
    (match eval_dyn env e1, eval_dyn env e2 with
     | Int n1, Int n2 -> Int (n1 + n2)
     | _ -> raise (TypeError "+ expects integers"))
  | Let (x, e1, e2) ->
    let v1 = eval_dyn env e1 in
    eval_dyn (extend env x v1) e2
  | Eq0 e1 ->
    (match eval_dyn env e1 with
     | Int n -> Bool (n = 0)
     | _ -> raise (TypeError "eq0 expects an integer"))
  | If (e1, e2, e3) ->
    (match eval_dyn env e1 with
     | Bool b -> if b then eval_dyn env e2 else eval_dyn env e3
     | _ -> raise (TypeError "if expects a boolean"))
  | Fun (x, body) -> Closure ([], x, body)           (* ignore A: (•, λx.e) *)
  | Call (e1, e2) ->
    (match eval_dyn env e1 with
     | Closure (_, x, body) ->
       let v1 = eval_dyn env e2 in
       eval_dyn (extend env x v1) body              (* current A, x:v1 *)
     | _ -> raise (TypeError "call of a non-function"));;
```

Out[42]:
```
val eval_dyn : env -> exp -> value = <fun>
```

Static vs. Dynamic Scoping

```
let x = 1 in
let f = fun y -> x + y in
let x = 10 in
f 5
```

Which `x` does the body of `f` see? First, ask real OCaml:

In [43]:
```
let x = 1 in
let f = fun y -> x + y in
let x = 10 in
f 5;;
```

```
File "[43]", line 3, characters 4-5:
3 | let x = 10 in
        ^
Warning 26 [unused-var]: unused variable x.
```

Out[43]:
```
- : int = 6
```

`6`: OCaml is statically scoped. OCaml even warns that the second `x` is unused — no code can see it, because `f`'s body refers to the `x` that was in scope where `f` was defined.

In [44]:
```
let prog =
  Let ("x", Val (Int 1),
  Let ("f", Fun ("y", Plus (Ident "x", Ident "y")),
  Let ("x", Val (Int 10),
  Call (Ident "f", Val (Int 5)))));;

eval [] prog;;          (* static: x is 1 where f was defined *)
eval_dyn [] prog;;      (* dynamic: x is 10 where f is called *)
```

Out[44]:
```
val prog : exp =
  Let ("x", Val (Int 1),
   Let ("f", Fun ("y", Plus (Ident "x", Ident "y")),
    Let ("x", Val (Int 10), Call (Ident "f", Val (Int 5)))))
```

Out[44]:
```
- : value = Int 6
```

Out[44]:
```
- : value = Int 15
```

Dynamic scoping also breaks functions that return functions: `add 3` should remember `x = 3`, but a dynamically scoped closure throws its environment away.

```
let add = fun x -> fun y -> x + y in
let add3 = add 3 in
add3 4
```

In [45]:
```
let curried =
  Let ("add", Fun ("x", Fun ("y", Plus (Ident "x", Ident "y"))),
  Let ("add3", Call (Ident "add", Val (Int 3)),
  Call (Ident "add3", Val (Int 4))));;

eval [] curried;;
```

Out[45]:
```
val curried : exp =
  Let ("add", Fun ("x", Fun ("y", Plus (Ident "x", Ident "y"))),
   Let ("add3", Call (Ident "add", Val (Int 3)),
    Call (Ident "add3", Val (Int 4))))
```

Out[45]:
```
- : value = Int 7
```

In [46]:
```
eval_dyn [] curried;;     (* Failure: x is undefined when add3's body runs *
```

```
Exception: Failure "undefined: x".

Raised at Stdlib.failwith in file "stdlib.ml", line 29, characters 17-33
Called from eval_dyn in file "[42]", line 6, characters 11-26
Called from <unknown> in file "[46]", line 1, characters 0-19
Called from Topeval.load_lambda in file "toplevel/byte/topeval.ml", line 93,
characters 4-14
```

Scaling Up

- Operational semantics 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

Like Lego: a handful of small rules, snapped together, describe a whole language.


LEGO City

Summary

- **Operational semantics** defines what programs mean with inference **rules** for a judgment $e \Rightarrow v$
- A **derivation** is a proof tree built from the rules; e means v exactly when a derivation of $e \Rightarrow v$ exists
- A **definitional interpreter** has one `match` case per rule, and its recursive call tree has the same shape as the derivation
- Variables can be handled by **substitution** $e\{v/x\}$ or by an **environment** $A;\ e \Rightarrow v$ (an association list in OCaml)
- Functions evaluate to **closures** $(A, \lambda x.\, e)$; capturing the defining environment gives **static scoping**, using the caller's environment gives **dynamic scoping**