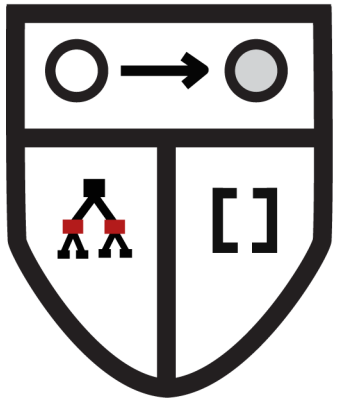


# CS 3110

## The Environment Model

Prof. Clarkson

Fall 2019

Today's music: Selections from *Doctor Who* soundtracks by Murray Gold

# CLICKER QUESTION 1

# Review

**Previously in 3110:**

- Interpreters
- Substitution model

**Today:**

- Small-step vs. big-step evaluation
- Environment model
- Dynamic vs. static scope

# Small-step evaluation

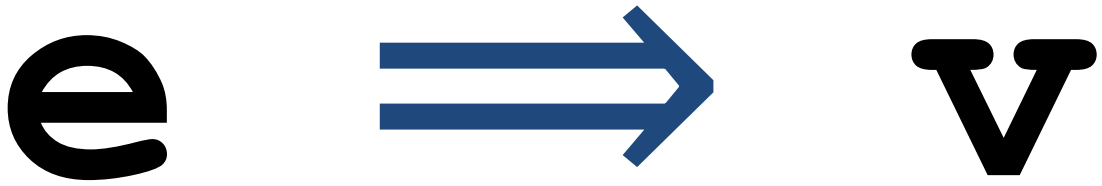
Small (single) step relation: $e \longrightarrow e'$

- $(10 + 1) + (5 + 6) \longrightarrow 11 + (5 + 6)$
- $11 + (5 + 6) \longrightarrow 11 + 11$
- $11 + 11 \longrightarrow 22$
- $22 \not\longrightarrow$

Multistep relation: $e \longrightarrow^* e'$

- $(10 + 1) + (5 + 6) \longrightarrow^* (10 + 1) + (5 + 6)$
- $(10 + 1) + (5 + 6) \longrightarrow^* 11 + (5 + 6)$
- $(10 + 1) + (5 + 6) \longrightarrow^* 11 + 11$
- $(10 + 1) + (5 + 6) \longrightarrow^* 22$

The final value is what OCaml actually gives us…




big-step relation

the `eval` function we implemented

# Big-step evaluation


$$e \Longrightarrow v$$

**Big and small should be consistent:**

for all expressions **e** and values **v**,

**e** $\Longrightarrow$ **v** if and only if **e** $\rightarrow^*$ **v**

# BIG-STEP SEMANTICS

# SimPL

```
e ::= x | i | b
    | e1 bop e2
    | let x = e1 in e2
    | if e1 then e2 else e3

bop ::= + | * | <=
```

# Semantics

`v` $\Longrightarrow$ `v`

`e1 + e2` $\Longrightarrow$ `v`

*if* `e1` $\Longrightarrow$ `v1`

*and* `e2` $\Longrightarrow$ `v2`

*and* `v` *is the result of primitive operation* `v1 + v2`

`x` $\not\Longrightarrow$

# Semantics

```
let x = e1 in e2 ⟹ v2
  if e1 ⟹ v1
  and e2{v1/x} ⟹ v2

if e1 then e2 else e3 ⟹ v2
  if e1 ⟹ true
  and e2 ⟹ v2

if e1 then e2 else e3 ⟹ v3
  if e1 ⟹ false
  and e3 ⟹ v3
```

Demo

# ENVIRONMENT MODEL

# Substitution could be slow

```
let x = 0 in 42
```

Imagine a large block of code instead of just [42]

```
let b = ... in
let x = ... in
if b then x + 1 else x - 1
```

Imagine large blocks of code instead of just [x+1] and [x-1]

# Dictionaries are fast

```
let x = 42 in
```

Dynamic environment

| x | 42 |
|---|---|

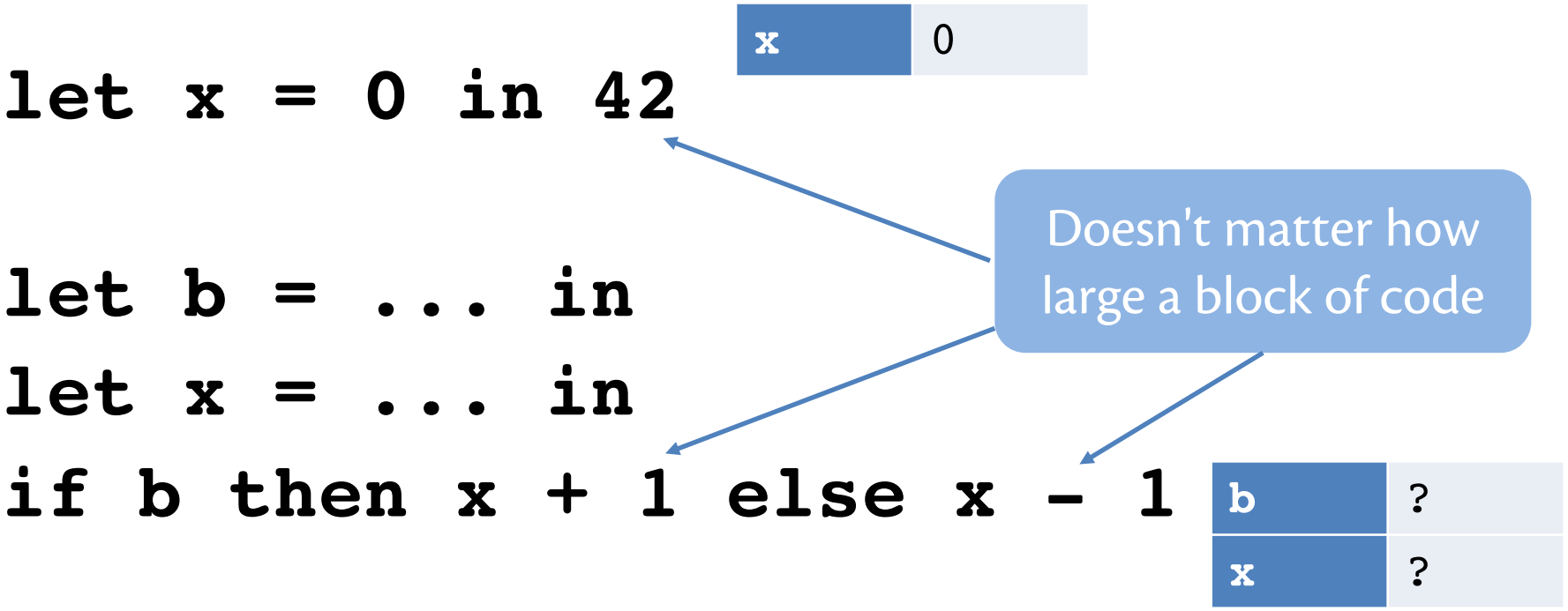
```
let y = 3110 in
```

```
x + y
```

| x | 42 |
|---|---|
| y | 3110 |

# Dynamic environment

- Maps variable names to values in current scope
- Implements a kind of lazy substitution



machine configuration


environment-model big-step relation

# Values

$\langle \texttt{env}, \texttt{v} \rangle \Longrightarrow \texttt{v}$

# Binary operators

$\langle$`env, v`$\rangle \Longrightarrow$ `v`

$\langle$`env, e1 + e2`$\rangle \Longrightarrow$ `v`

*if* $\langle$`env, e1`$\rangle \Longrightarrow$ `v1`

*and* $\langle$`env, e2`$\rangle \Longrightarrow$ `v2`

*and* **`v`** *is the result of primitive operation* **`v1 + v2`**

# Let

$$\langle env,\ \texttt{let}\ x = e1\ \texttt{in}\ e2\rangle \Longrightarrow v2$$

$$\quad \texttt{if}\ \langle env,\ e1\rangle \Longrightarrow v1$$

$$\quad \texttt{and}\ \langle env[x \mapsto v1],\ e2\rangle \Longrightarrow v2$$

think of this as recording the substitution in case it is ever needed

**env[x ↦ v]:** **env** with **x** bound to **v**

# Variables

$\langle$`env, x`$\rangle \Longrightarrow$ `v`

*if* `v = env(x)`

**`env(x)`** : the value to which **`env`** binds **`x`**

# Function values v1.0

Since functions are values:

$\langle$`env, fun x -> e`$\rangle \Longrightarrow$ `fun x -> e`

# Function application rule v1.0

⟨env,e1 e2⟩ ⟹ v

if ⟨env,e1⟩ ⟹ fun x -> e

and ⟨env,e2⟩ ⟹ v2

and ⟨env[x ↦ v2],e⟩ ⟹ v

# CLICKER QUESTION 2

# Scope: OCaml

What does OCaml say this evaluates to?

```
let x = 1 in
let f = fun y -> x in
let x = 2 in
   f 0
- : int = 1
```

# Scope: our semantics

What does our semantics say?

```
let x = 1 in
{x:1} let f = fun y -> x in
{x:1,f:(fun y -> x)} let x = 2 in
  {x:2,f:(fun y -> x)} f 0
```

⟨`{x:2,f:(fun y -> x)}, f 0`⟩ ⟹ `???`

1. Evaluate `f` to a value: `fun y -> x`
2. Evaluate `0` to a value: `0`
3. Extend environment to map parameter:
   `{x:2, f:(fun y -> x), y:0}`
4. Evaluate body `x` in that environment
5. Return `2`

`2 <> 1`

# Why different answers?

Two different rules for variable scope:

- Rule of dynamic scope (our semantics so far)
- Rule of lexical scope (OCaml)

# Dynamic scope

**Rule of dynamic scope:** The body of a function is evaluated in the current dynamic environment at the time the function is **called**, not the old dynamic environment that existed at the time the function was defined.

- Causes our semantics to use latest binding of **x**
- Thus return **2**

# Lexical scope

**Rule of lexical scope:** The body of a function is evaluated in the old dynamic environment that existed at the time the function was **defined**, not the current environment when the function is called.

- Causes OCaml to use earlier binding of **x**
- Thus return **1**

# Lexical scope

**Rule of** n is evaluat that existed , not the curr is called.

- Cause
- Thus

# Implementing time travel

**Q:** How can functions be evaluated in old environments?

**A:** The language implementation keeps old environments around as necessary

# Implementing time travel

A function value is really a data structure called a function closure that has two parts:

- The code, an expression **`e`**
- The environment **`env`** that was current when the function was defined
- We'll notate that data structure as **`(|e , env|)`**

**`(|e , env|)`** is like a pair

- But indivisible: you cannot write OCaml syntax to access the pieces
- And inexpressible: you cannot directly write it in OCaml syntax

# Closures in OCaml bytecode compiler

https://github.com/ocaml/ocaml/search?q=kclosure

**Results in ocaml/ocaml**

bytecomp/instruct.ml — OCaml

Showing the top match  Last indexed on Sep 15, 2016

```
63    | Krestart
64    | Kgrab of int                    (* number of arguments *)
65    | Kclosure of label * int
```

bytecomp/printinstr.ml — OCaml

Showing the top match  Last indexed on Sep 15, 2016

```
35    | Kgrab n -> fprintf ppf "\tgrab %i" n
36    | Kclosure(lbl, n) ->
37        fprintf ppf "\tclosure L%i, %i" lbl n
```

bytecomp/instruct.mli — OCaml

Showing the top match  Last indexed on Aug 10

```
84    | Kgrab of int                    (* number of arguments *)
85    | Kclosure of label * int
86    | Kclosurerec of label list * int
87    | Koffsetclosure of int
```

# Function application rule v2.0

⟨env, e1 e2⟩ ⟹ v

if ⟨env, e1⟩ ⟹
(|fun x -> e , defenv|)

and ⟨env, e2⟩ ⟹ v2

and ⟨defenv[x ↦ v2], e⟩ ⟹ v

# Function values v2.0

Anonymous functions **`fun x -> e`** are closures:

⟨`env, fun x -> e`⟩

⟹ `(|fun x -> e , env|)`

Demo

# Lexical vs. dynamic scope

- Consensus after decades of programming language design is that **lexical scope is the right choice**
  - it supports the Principle of Name Irrelevance: name of variable shouldn't matter to meaning of program
  - programmers free to change names of local variables
  - type checker can prevent more run-time errors
- Dynamic scope is useful in some situations
  - Some languages use it as the norm (e.g., Emacs LISP, LaTeX)
  - Some languages have special ways to do it (e.g., Perl, Racket)
  - But most languages just don't have it
- Exception handling resembles dynamic scope:
  - `raise e` transfers control to the "most recent" exception handler
  - like how dynamic scope uses "most recent" binding of variable

# Upcoming events

- [last night]: R8 due (last reflection!)
- [Tue/Wed]: MS1 demos in section
- [Thur]: MS1 due in CMS

*This is closure.*

**THIS IS 3110**