Programming Languages and Compilers (CS 421)

Elsa L Gunter

2112 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2017/CS421D

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Local Variable Creation

// ρ_3 = {test → 3.7, a → 1, b → 5}

# let b = 5 * 4

// ρ_4 = {b → 20, test → 3.7, a → 1}

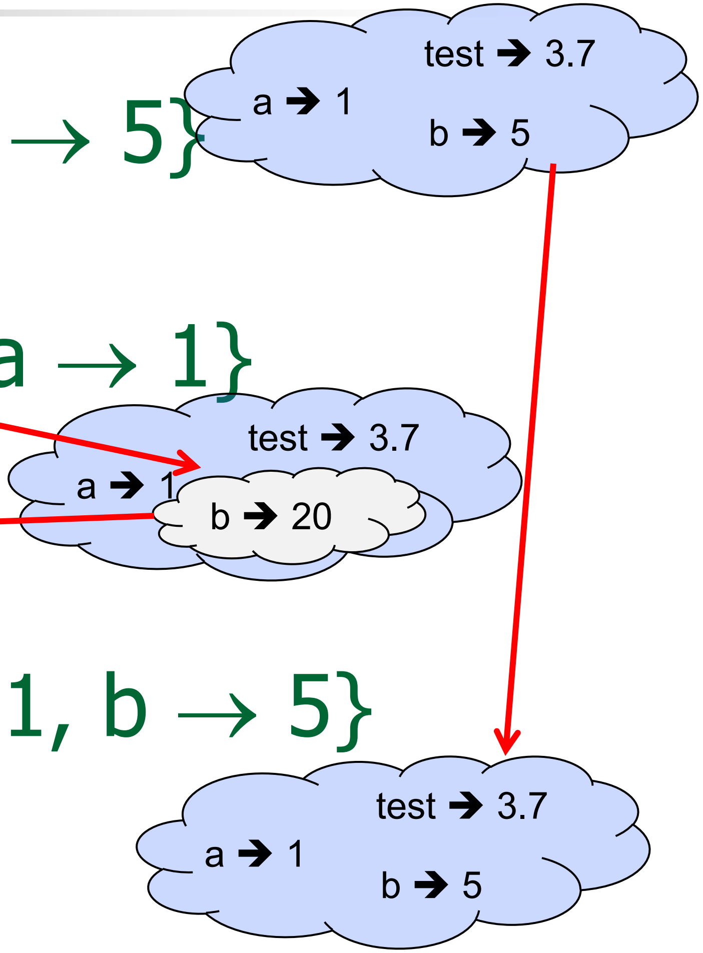
    in 2 * b;;

- : int = 40

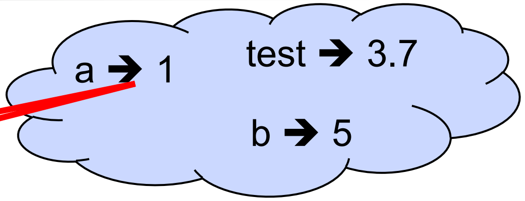
// ρ_5 = ρ_3= {test → 3.7, a → 1, b → 5}

b;;

- : int = 5



Local let binding



// ρ_5 = {test → 3.7, a → 1, b → 5}

# let c =

    let b = a + a

// ρ_6 = {b → 2} + ρ_3

//   ={b → 2, test → 3.7, a → 1}

    in b * b;;

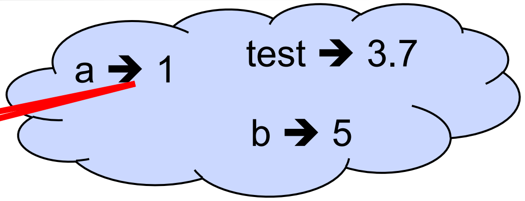
val c : int = 4

// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}

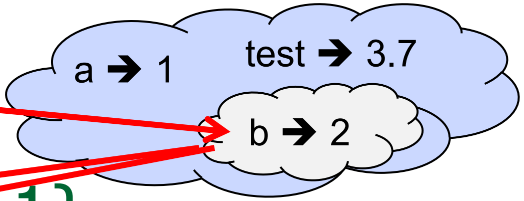
# b;;

- : int = 5

Local let binding



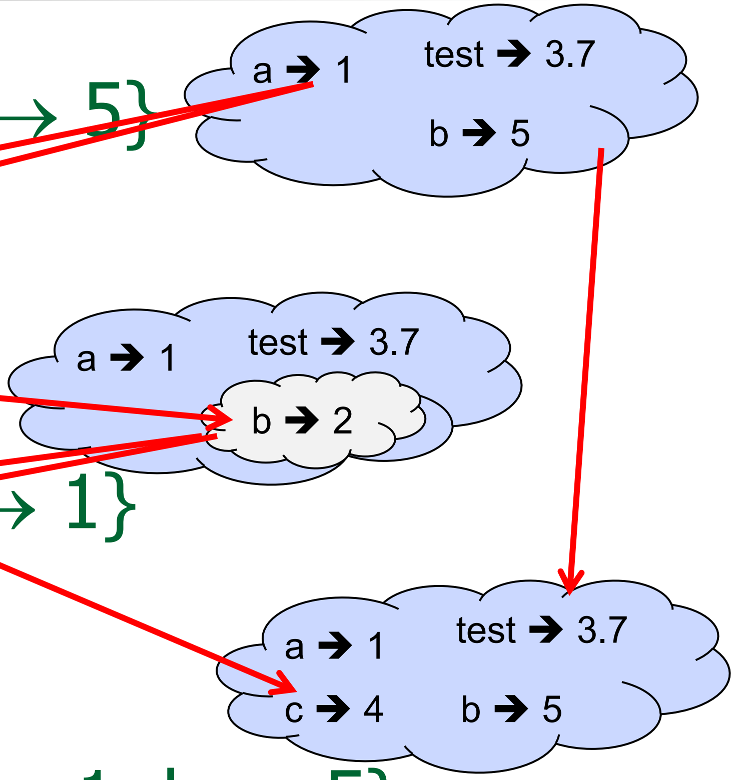
// ρ_5 = {test → 3.7, a → 1, b → 5}
# let c =
    let b = a + a



// ρ_6 = {b → 2} + ρ_3
//   ={b → 2, test → 3.7, a → 1}
    in b * b;;
val c : int = 4
// ρ_7 = {c → 4, test → 3.7, a → 1, b → 5}
# b;;
- : int = 5

Local let binding

```
//  ρ5 = {test → 3.7, a → 1, b → 5}
# let c =
    let b = a + a
//  ρ6 = {b → 2} + ρ3
//       ={b → 2, test → 3.7, a → 1}
    in b * b;;
val c : int = 4
//  ρ7 = {c → 4, test → 3.7, a → 1, b → 5}
# b;;
- : int = 5
```



Booleans (aka Truth Values)

```
# true;;
- : bool = true
# false;;
- : bool = false
```

// $\rho_7 = \{c \rightarrow 4, test \rightarrow 3.7, a \rightarrow 1, b \rightarrow 5\}$

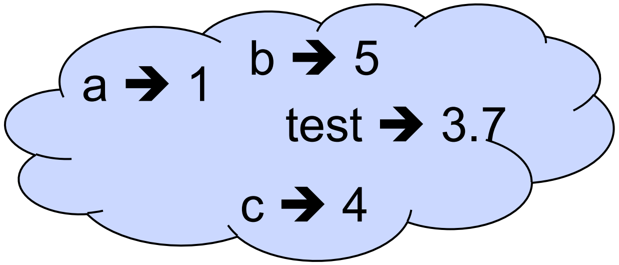
```
# if b > a then 25 else 0;;
- : int = 25
```

Booleans and Short-Circuit Evaluation

```
# 3 > 1 && 4 > 6;;
- : bool = false
# 3 > 1 || 4 > 6;;
- : bool = true
# (print_string "Hi\n"; 3 > 1) || 4 > 6;;
Hi
- : bool = true
# 3 > 1 || (print_string "Bye\n"; 4 > 6);;
- : bool = true
# not (4 > 6);;
- : bool = true
```

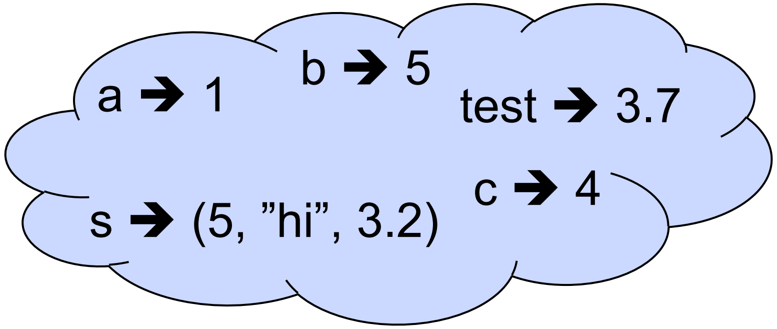
Tuples as Values

// ρ_7 = {c → 4, test → 3.7,
a → 1, b → 5}


a → 1
b → 5
test → 3.7
c → 4

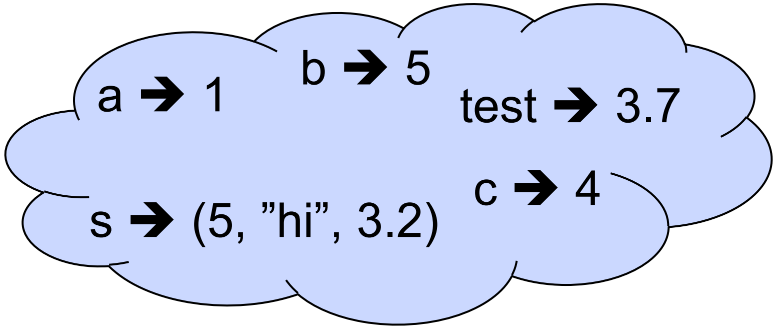
```
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// ρ_8 = {s → (5, "hi", 3.2),
c → 4, test → 3.7,
a → 1, b → 5}

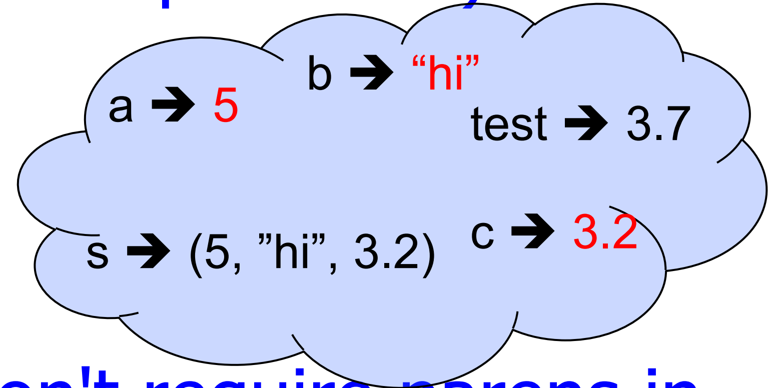

a → 1
b → 5
test → 3.7
c → 4
s → (5, ”hi”, 3.2)

Pattern Matching with Tuples

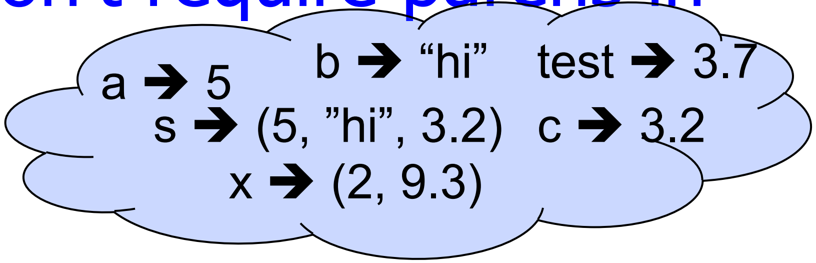
/ ρ_8 = {s → (5, "hi", 3.2),
c → 4, test → 3.7,
a → 1, b → 5}


a → 1
b → 5
test → 3.7
s → (5, ”hi”, 3.2)
c → 4

```
# let (a,b,c) = s;;  (* (a,b,c) is a pattern *)
val a : int = 5
val b : string = "hi"
val c : float = 3.2
```


a → 5
b → “hi”
test → 3.7
s → (5, ”hi”, 3.2)
c → 3.2

```
# let x = 2, 9.3;; (* tuples don't require parens in
  Ocaml *)
val x : int * float = (2, 9.3)
```


a → 5
b → “hi”
test → 3.7
s → (5, ”hi”, 3.2)
c → 3.2
x → (2, 9.3)

Nested Tuples

```
#  (*Tuples can be nested *)
let d = ((1,4,62),("bye",15),73.95);;
val d : (int * int * int) * (string * int) * float =
  ((1, 4, 62), ("bye", 15), 73.95)
#  (*Patterns can be nested *)
let (p,(st,_),_) = d;; (* _ matches all, binds nothing
   *)
val p : int * int * int = (1, 4, 62)
val st : string = "bye"
```

Functions on tuples

```
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Functions on tuples

```
# let plus_pair (n,m) = n + m;;
val plus_pair : int * int -> int = <fun>
# plus_pair (3,4);;
- : int = 7
# let double x = (x,x);;
val double : 'a -> 'a * 'a = <fun>
# double 3;;
- : int * int = (3, 3)
# double "hi";;
- : string * string = ("hi", "hi")
```

Save the Environment!

- A *closure* is a pair of an environment and an association of a pattern (e.g. (v1,…,vn) giving the input variables) with an expression (the function body), written:

$$< (v1,\ldots,vn) \rightarrow exp, \rho >$$

- Where ρ is the environment in effect when the function is defined (for a simple function)

Closure for plus_x

- When plus_x was defined, had environment:

$$\rho_{plus\_x} = \{\ldots, x \to 12, \ldots\}$$

- Recall: let plus_x y = y + x

  is really let plus_x = fun y -> y + x

- Closure for fun y -> y + x:

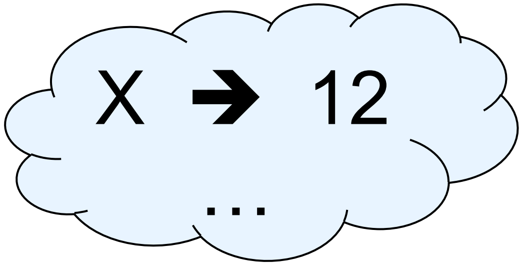
$$<y \to y + x, \rho_{plus\_x}>$$

- Environment just after plus_x defined:

$$\{plus\_x \to <y \to y + x, \rho_{plus\_x}>\} + \rho_{plus\_x}$$

Recall: let plus_x = fun x => y + x

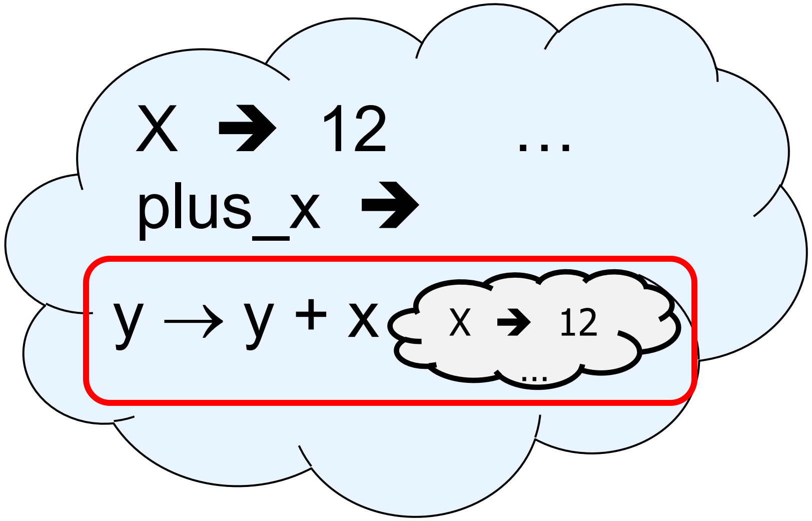
let x = 12



X ➔ 12

…

let plus_x = fun y => y + x



X ➔ 12 …

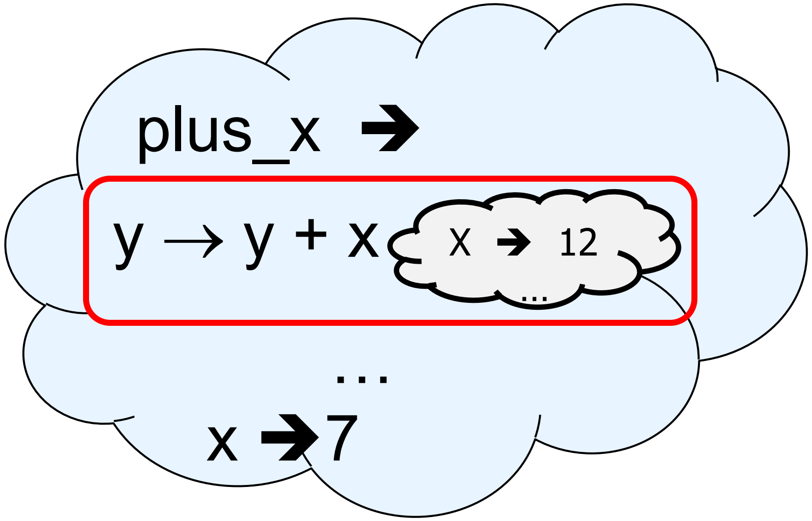
plus_x ➔

y → y + x

X ➔ 12

…

let x = 7



plus_x ➔

y → y + x

X ➔ 12

…

…

x ➔7

Closure for plus_pair

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before plus_pair defined
- Closure for fun (n,m) -> n + m:

$$<(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}}>$$

- Environment just after plus_pair defined:

$$\{plus\_pair \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus\_pair}} >\}$$

$$+ \rho_{\text{plus\_pair}}$$

Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
# let t = add_three 6 3 2;;
val t : int = 11
# let add_three =
   fun x -> (fun y -> (fun z -> x + y + z));;
val add_three : int -> int -> int -> int = <fun>
```

Again, first syntactic sugar for second

Curried vs Uncurried

- Recall

```
val add_three : int -> int -> int -> int = <fun>
```

- How does it differ from

```
# let add_triple (u,v,w) = u + v + w;;
val add_triple : int * int * int -> int = <fun>
```

- add_three is *curried*;
- add_triple is *uncurried*

Curried vs Uncurried

```
# add_triple (6,3,2);;
- : int = 11
# add_triple 5 4;;
Characters 0-10:
  add_triple 5 4;;
  ^^^^^^^^^^^
This function is applied to too many arguments,
maybe you forgot a `;'
# fun x -> add_triple (5,4,x);;
: int -> int = <fun>
```

Partial application of functions

```
let add_three x y z = x + y + z;;
```

```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
-  : int = 16
```

- Partial application also called *sectioning*

Functions with more than one argument

```
# let add_three x y z = x + y + z;;
val add_three : int -> int -> int -> int = <fun>
```

- What is the value of add_three?
- Let ρ_{add_three} be the environment before the declaration
- Remember:

```
let add_three =
 fun x -> (fun y -> (fun z -> x + y + z));;
```

Value: <x ->fun y -> (fun z -> x + y + z), ρ_{add_three} >

Match Expressions

```
# let triple_to_pair triple =
  match triple
  with (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);;
val triple_to_pair : int * int * int -> int * int =
   <fun>
```

- Each clause: pattern on left, expression on right
- Each x, y has scope of only its clause
- Use first matching clause

Recursive Functions

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
  val factorial : int -> int = <fun>
# factorial 5;;
- : int = 120
# (* rec  is needed for recursive function
   declarations *)
```

Recursion Example

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# let rec nthsq n =          (* rec for recursion *)
   match n                 (* pattern matching for cases *)
   with 0 -> 0               (* base case *)
   | n -> (2 * n -1)         (* recursive case *)
         + nthsq (n -1);;   (* recursive call *)
val nthsq : int -> int = <fun>
# nthsq 3;;
-   : int = 9
```

Structure of recursion similar to inductive proof

Recursion and Induction

```
# let rec nthsq n = match n with 0 -> 0
       | n -> (2 * n - 1) + nthsq (n - 1) ;;
```

- Base case is the last case; it stops the computation
- Recursive call must be to arguments that are somehow smaller - must progress to base case
- **if** or **match** must contain base case
- Failure of these may cause failure of termination

Functions as arguments

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
# let g = thrice plus_two;;
val g : int -> int = <fun>
# g 4;;
- : int = 10
# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
- : string = "Hi! Hi! Hi! Good-bye!"
```

Higher Order Functions

- A function is *higher-order* if it takes a function as an argument or returns one as a result
- Example:

```
# let compose f g = fun x -> f (g x);;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c ->
  'b = <fun>
```

- The type ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b is a higher order type because of ('a -> 'b) and ('c -> 'a) and -> 'c -> 'b

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

Thrice

- Recall:

```
# let thrice f x = f (f (f x));;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- How do you write thrice with compose?

```
# let thrice f = compose f (compose f f);;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
```

- Is this the only way?

Lambda Lifting

- You must remember the rules for evaluation when you use partial application

```
# let add_two = (+) (print_string "test\n"; 2);;
test
val add_two : int -> int = <fun>
# let add2 =      (* lambda lifted *)
   fun x -> (+) (print_string "test\n"; 2) x;;
val add2 : int -> int = <fun>
```

Lambda Lifting

```
# thrice add_two 5;;
- : int = 11
# thrice add2 5;;
test
test
test
- : int = 11
```

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied

Evaluating declarations

- Evaluation uses an environment ρ
- To evaluate a (simple) declaration let x = e
  - Evaluate expression e in ρ to value v
  - Update ρ with x v: $\{x \rightarrow v\} + \rho$

- Update: $\rho_1 + \rho_2$ has all the bindings in ρ_1 and all those in ρ_2 that are not rebound in ρ_1

$\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\}$

$= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$

Evaluating expressions

- Evaluation uses an environment ρ
- A constant evaluates to itself
- To evaluate an variable, look it up in ρ: $\rho(v)$
- To evaluate uses of +, _ , etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: let x = e1 in e2
  - Eval e1 to v, eval e2 using $\{x \rightarrow v\} + \rho$

Evaluating conditions expressions

- To evaluate a conditional expression:
  if b then e1 else e2
  - Evaluate b to a value v
  - If v is True, evaluate e1
  - If v is False, evaluate e2

Evaluation of Application with Closures

- Given application expression $f(e_1,\ldots,e_n)$
- Evaluate $(e_1,\ldots,e_n)$ to value $(v_1,\ldots,v_n)$
- In environment ρ, evaluate left term to closure, $c = <(x_1,\ldots,x_n) \rightarrow b, \rho'>$
  - $(x_1,\ldots,x_n)$ variables in (first) argument
- Update the environment ρ' to

  $\rho'' = \{x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n\}+ \rho'$
- Evaluate body b in environment ρ''

Evaluation of Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \rightarrow <y \rightarrow y + x, \rho_{plus\_x}>, \ldots,$
  $y \rightarrow 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \rightarrow 12, \ldots, y \rightarrow 24, \ldots\}$
- Eval (plus_x y, ρ) rewrites to
- App (Eval(plus_x, ρ) , Eval(y, ρ)) rewrites to
- App (Eval(plus_x, ρ) , 3) rewrites to
- App ($<y \rightarrow y + x, \rho_{plus\_x}>$, 3) rewrites to

  …

Evaluation of Application of plus_x;;

- Have environment:

  $\rho = \{plus\_x \to <y \to y + x, \rho_{plus\_x} >, \ldots ,$
  $y \to 3, \ldots\}$

  where $\rho_{plus\_x} = \{x \to 12, \ldots , y \to 24, \ldots\}$
- $App\ (<y \to y + x, \rho_{plus\_x} >, 3)$ rewrites to
- $Eval\ (y + x, \{y \to 3\} +\rho_{plus\_x})$ rewrites to
- $Eval\ (y, \{y \to 3\} +\rho_{plus\_x}) +$
  $Eval\ (x, \{y \to 3\} +\rho_{plus\_x})$ rewrites to
- $Eval\ (y, \{y \to 3\} +\rho_{plus\_x}) + 12$ rewrites to
- $3+ 12 = 15$

Evaluation of Application of plus_pair

- Assume environment

$\rho = \{x \to 3\ldots,$

$\quad$ plus_pair $\to <(n,m) \to n + m, \rho_{\text{plus\_pair}}>\} +$

$\quad \rho_{\text{plus\_pair}}$

- Eval (plus_pair (4,x), ρ)=
- App (Eval (plus_pair, ρ), Eval ((4,x), ρ)) =
- App ($<(n,m) \to n + m, \rho_{\text{plus\_pair}}>$, (4,3)) =
- Eval (n + m, {n -> 4, m -> 3} + $\rho_{\text{plus\_pair}}$) =
- Eval (4 + 3, {n -> 4, m -> 3} + $\rho_{\text{plus\_pair}}$) = 7

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun n -> n + 5;;
(* 0 *)
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 0 *)?

Answer

let f = fun n -> n + 5;;

$\rho_0 = \{f \rightarrow <n \rightarrow n + 5, \{\ \}>\}$

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
(* 1 *)
let f = pair_map f;;
let a = f (4,6);;
```

What is the environment at (* 1 *)?

Answer

ρ_0 = {f → <n → n + 5, { }>}

let pair_map g (n,m) = (g n, g m);;

ρ_1 = {pair_map →

<g → fun (n,m) -> (g n, g m),

{f → <n → n + 5, { }>}>,

f → <n → n + 5, { }>}

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
(* 2 *)
let a = f (4,6);;
```

What is the environment at (* 2 *)?

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}

ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}

let f = pair_map f;;

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}

ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}

Eval(pair_map f, ρ_1) =

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}

ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
        f→<n → n + 5, { }>}

Eval(pair_map f, ρ_1) =

App (<g→fun (n,m) -> (g n, g m), ρ_0>,
        <n → n + 5, { }>) =

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}

ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}

Eval(pair_map f, ρ_1) =

App (<g→fun (n,m) -> (g n, g m), ρ_0>,
<n → n + 5, { }>) =

Eval(fun (n,m)->(g n, g m), {g→<n→n + 5, { }>}+ρ_0)

=

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}

ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
f→<n → n + 5, { }>}

Eval(pair_map f, ρ_1) =

App (<g→fun (n,m) -> (g n, g m), ρ_0>,
<n → n + 5, { }>) =

Eval(fun (n,m)->(g n, g m), {g→<n→n + 5, { }>}+ρ_0)

=<(n,m) →(g n, g m), {g→<n→n + 5, { }>}+ρ_0>

=

Evaluate pair_map f

ρ_0 = {f → <n → n + 5, { }>}

ρ_1 = {pair_map →<g→fun (n,m) -> (g n, g m), ρ_0>,
    f→<n → n + 5, { }>}

Eval(pair_map f, ρ_1) =

App (<g→fun (n,m) -> (g n, g m), ρ_0>,
    <n → n + 5, { }>) =

Eval(fun (n,m)->(g n, g m), {g→<n→n + 5, { }>}+ρ_0)

=<(n,m) →(g n, g m), {g→<n→n + 5, { }>}+ρ_0>

=<(n,m) →(g n, g m), {g→<n→n + 5, { }>
    f→<n→n + 5, { }>}

Answer

ρ_1 = {pair_map →
<g → fun (n,m) -> (g n, g m),{f → <n → n + 5, { }>}>,
f → <n → n + 5, { }>}

let f = pair_map f;;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

Closure question

- If we start in an empty environment, and we execute:

```
let f = fun => n + 5;;
let pair_map g (n,m) = (g n, g m);;
let f = pair_map f;;
let a = f (4,6);;
```

(* 3 *)

What is the environment at (* 3 *)?

Final Evalution?

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

let a = f (4,6);;

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

Eval(f (4,6), ρ_2) =

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),

{g → <n → n + 5, { }>,

f → <n → n + 5, { }>}>,

pair_map → <g → fun (n,m) -> (g n, g m),

{f → <n → n + 5, { }>}>}

Eval(f (4,6), ρ_2) = App(Eval(f, ρ_2), Eval((4,6), ρ_2) =

Evaluate f (4,6);;

ρ_2 = {f → <(n,m) →(g n, g m),
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
pair_map → <g → fun (n,m) -> (g n, g m),
{f → <n → n + 5, { }>}>}

Eval(f (4,6), ρ_2) = App(Eval(f, ρ_2), Eval((4,6), ρ_2) =
App(<(n,m) →(g n, g m), {g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
(4,6)) =

Evaluate f (4,6);;

App(<(n,m) →(g n, g m), {g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
(4,6)) =

Eval((g n, g m), {n → 4, m → 6} +
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) =

Evaluate f (4,6);;

App(<(n,m) →(g n, g m), {g → <n → n + 5, { }>,
f → <n → n + 5, { }>}>,
(4,6)) =

Eval((g n, g m), {n → 4, m → 6} +
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) =

(Eval(g n, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) ,
Eval(g m, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>})) =

Evaluate f (4,6);;

(Eval(g n, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) ,
Eval(g m, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>})) =
(App(Eval(g {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) ,
Eval(n, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>})),
App(Eval(g {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) ,
Eval(n, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}))) =

Evaluate f (4,6);;

(App(Eval(g {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) ,
Eval(n, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>})),
App(Eval(g {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}) ,
Eval(n, {n → 4, m → 6, g → <n → n + 5, { }>,
f → <n → n + 5, { }>}))) =
(App(<n → n + 5, { }>, 4),
App (<n → n + 5, { }>, 6)) =

Evaluate f (4,6);;

(App(<n → n + 5, { }>, 4),

App (<n → n + 5, { }>, 6)) =

(Eval(n + 5, {n → 4} + { }),

Eval(n + 5, {n → 6} + { })) =

Evaluate f (4,6);;

(App(<n → n + 5, { }>, 4),
 App (<n → n + 5, { }>, 6)) =

(Eval(n + 5, {n → 4} + { }),
 Eval(n + 5, {n → 6} + { })) =

(Eval(4 + 5, {n → 4}), Eval(6 + 5, {n → 6})) =

Evaluate f (4,6);;

(App(<n → n + 5, { }>, 4),
 App (<n → n + 5, { }>, 6)) =

(Eval(n + 5, {n → 4} + { }),
 Eval(n + 5, {n → 6} + { })) =

(Eval(4 + 5, {n → 4}), Eval(6 + 5, {n → 6})) =
 (9, 11)