Programming Languages and Compilers (CS 421)

#3: Closures, evaluation of function applications, order of evaluation

#4: Evaluation and Application rules using symbolic rewriting

Madhusudan Parthasarathy (madhu)

3112 Siebel Center madhu@illinois.edu

https://courses.engr.illinois.edu/cs421/fa2018/CS421D

Based on slides by Elsa Gunter, who made these slides mostly, in part borrowing from slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Logistics

- ML 1
  - Started today. Must be taken by Thursday.
  - Schedule and take it if you haven't already!
  - Many tries (but finitely many tries).
  - Idea: Test that you are indeed the person doing your homework!

- Practice ML1
  - Does NOT count against your grade
  - Prepares you for ML1. Take as many instances of it as you can.

- WA 1
  - Make sure you do one question (has a sequence of subquestions)
  - And after you are shown the score ("14/16", etc.), press the GRADE button

RECAP OF LAST LECTURE

Order of Evaluation

- Evaluating (f e1)
  - Evaluate e1 first
  - Then evaluate f on the value of e1

Booleans (aka Truth Values) Order of Evaluation

```
# 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
```

Functions as arguments and functions as return values

Partial application of functions

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

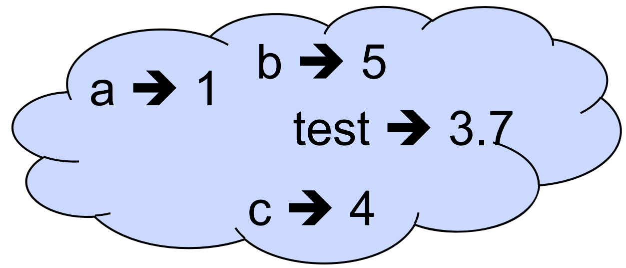
```
# let h = add_three 5 4;;
val h : int -> int = <fun>
# h 3;;
- : int = 12
# h 7;;
- : int = 16
```

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!"
```

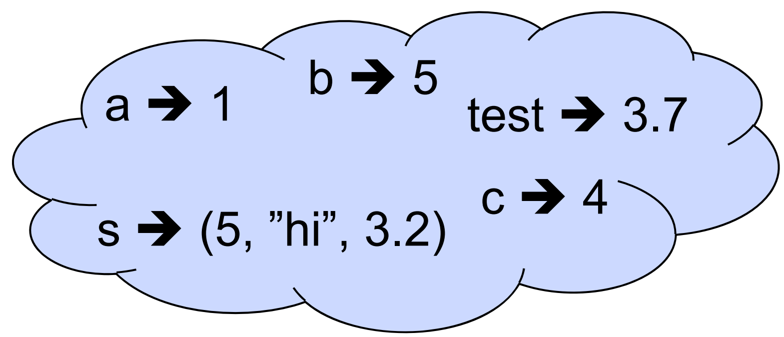
Tuples as Values

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



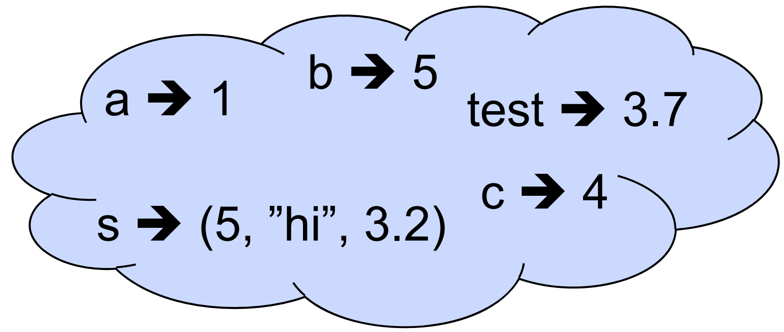

```
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)
```

// $\rho_8 = \{s \rightarrow (5, \text{"hi"}, 3.2),$
$c \rightarrow 4, \text{test} \rightarrow 3.7,$
$a \rightarrow 1, b \rightarrow 5\}$



Pattern Matching with Tuples

/ $\rho_8 = \{s \rightarrow (5, \text{"hi"}, 3.2),$
$c \rightarrow 4, test \rightarrow 3.7,$
$a \rightarrow 1, b \rightarrow 5\}$

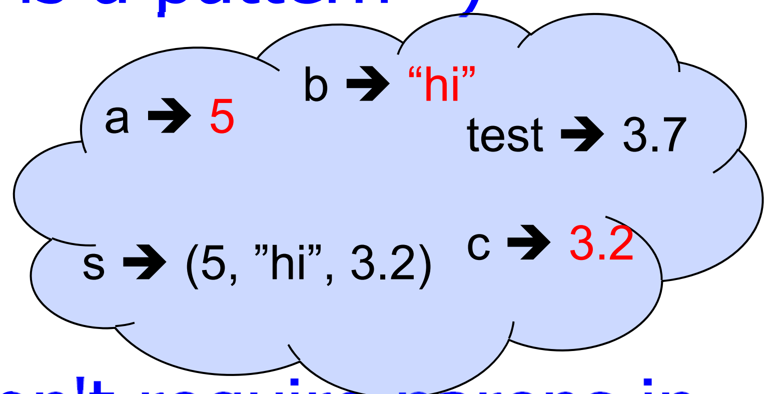


let (a,b,c) = s;; (* (a,b,c) is a pattern *)

val a : int = 5

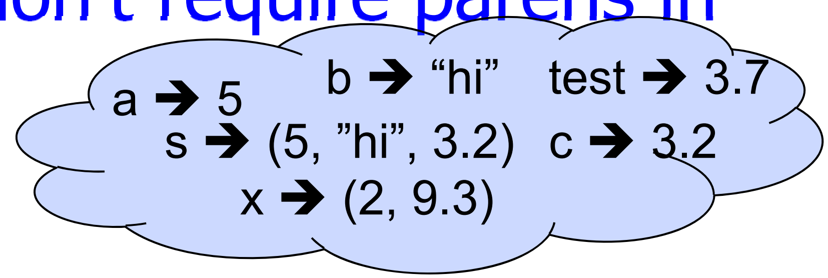
val b : string = "hi"

val c : float = 3.2



let x = 2, 9.3;; (* tuples don't require parens in Ocaml *)

val x : int * float = (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"
```

Save the Environment!

- A *closure* is a pair of an environment and an association of a sequence of variables (the input variables) with an expression (the function body), written:

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

- Where ρ_f is the environment in effect when f is defined (if f is a simple function)

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")
```

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

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

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

- Environment just after plus_pair defined:

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

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

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

Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

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

- Environment just after plus_pair defined:

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

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!"
```

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")
```

Recall closures: Closure for plus_pair

- Assume ρ_{plus_pair} was the environment just before plus_pair defined
- Closure for plus_pair:

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

- Environment just after plus_pair defined:

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

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 `;'
```

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*

Evaluation

$$\text{Exp} \xrightarrow[\text{Rewriting}]{\text{Evaluation}} \text{Value}$$

=

- How exactly do we evaluate expressions to values?

Values

Bool Int

Float Closures

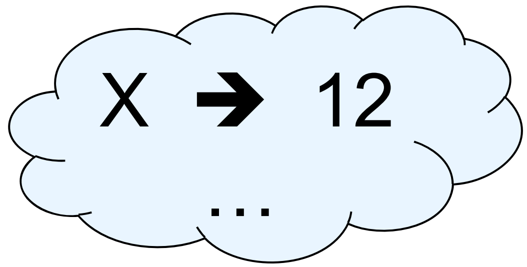
Strings

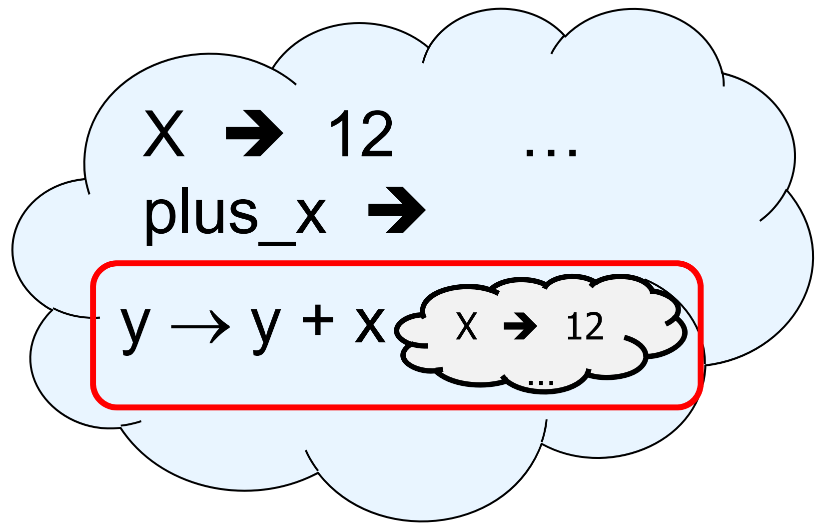
Tuples

Lists

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

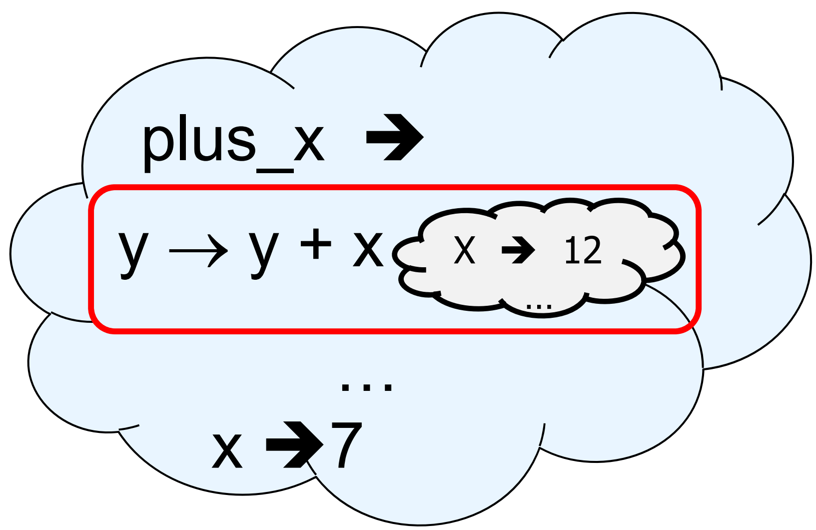
let x = 12





let plus_x = fun y => y + x

let x = 7



Recall closures: 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:

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

fn body env it was created in

- Environment just after plus_x defined:

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

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: $\{x \to v\} + \rho$

- 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

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

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

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 $\{x \to v\} + \rho$

More generally, let pattern = e1 in e2

Evaluating expressions

- To evaluate if e1 then e2 else e3 under an environment ρ
  Eval e1 using ρ
  If it evaluates to true, then eval e2 using ρ.
  Else, (i.e., it evaluates to false), then eval e3 using ρ.

- To evaluate match e with p1 -> e1 | p2 -> e2… | pn -> en in env ρ
  Evaluate e to v.
  If v matches p1, then eval e1, else
    if v matches p2, then eval e2, else …
      if v matches pn then eval en
  (all evaluations using environment

Evaluation of Application with Closures

- Given application expression $f(e_1,\ldots,e_n)$ in env ρ, where f is an identifier
- 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
- Evaluate $(e_1,\ldots,e_n)$ to value $(v_1,\ldots,v_n)$ using ρ
- 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:
  $\rho = \{plus\_x \to \langle y \to y + x, \rho_{plus\_x} \rangle, \ldots , y \to 3, \ldots\}$
  where $\rho_{plus\_x} = \{x \to 12, \ldots , y \to 24, \ldots\}$

- Eval (plus_x y, ρ) rewrites to

  App (Eval(plus_x, ρ) , Eval(y, ρ)) rewrites to

  App ($\langle y \to y + x, \rho_{plus\_x} \rangle$, 3) rewrites to

  Eval (y + x, $\{y \to 3\} + \rho_{plus\_x}$) rewrites to

  Eval (3 + 12 , ρ_{plus_x}) = 15

App

let add x y = x + y — fun

let add = fun x -> (fun y -> (x+y))

App (closure, value)

$\langle x \to (\text{fun } y \to (x+y)), \phi \rangle$

~~$\langle y \to 5+y, \phi \rangle$~~ 5

$$App(< x \ \to e, \rho >, v) \ = Eval\ (e, \{x \ \to v\} + \rho)$$

$App(\langle (x_1, \ldots x_m) \to e, \rho \rangle, \langle v_1, \ldots v_m \rangle) = Eval(e, \{x_1 \to v_1, \ldots x_m \to v_m\} + \rho)$

Note:

- App(closure, value) is independent of any external env as "value" has no variables to look up in external env
- The closure (or value, which in turn can be a closure) may have local environments that define variables it uses.
- So when applying a function, we "shut off the rest of the world" and application depends only on the given closure and value.
- Eval and App and mutually recursively defined. Eval calls App, App calls Eval.

$\langle y \to x+y, \{x \to 5\} \rangle$

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 $\rightarrow$ <n $\rightarrow$ n + 5, { }>}

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

ρ_0

ρ_1 = { pair_map $\rightarrow$

<g $\rightarrow$ fun (n,m) -> (g n, g m), {f $\rightarrow$ <n $\rightarrow$ n + 5, { }>}>,

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

I.e.

ρ_1 = {pair_map $\rightarrow$

<g $\rightarrow$ fun (n,m) -> (g n, g m), ρ_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);;
let f = pair_map f;;
(* 2 *)
let a = f (4,6);;
```

What is the environment at (* 2 *)?

Evaluate pair_map f

ρ_0 = {f $\rightarrow$ <n $\rightarrow$ n + 5, { }>}

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

let f = pair_map f;;

Evaluate pair_map f

ρ_0 = {f $\rightarrow$ <n $\rightarrow$ n + 5, { }>}

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

Eval(pair_map f, ρ_1) =

Evaluate pair_map f

$\rho_0 = \{f \rightarrow \langle n \rightarrow n + 5, \{\ \}\rangle\}$

$\rho_1 = \{\text{pair\_map} \rightarrow \langle g \rightarrow \text{fun } (n,m) \text{ -> } (g\ n, g\ m), \rho_0\rangle,$

$\qquad f \rightarrow \langle n \rightarrow n + 5, \{\ \}\rangle\}$

$\text{Eval}(\text{pair\_map } f, \rho_1)$

$= \text{App}(\text{Eval } (\text{pair\_map}, \rho_1), \text{Eval}(f, \rho_1))$

$= \text{App } (\langle g \rightarrow \underline{\text{fun } (n,m) \text{ -> } (g\ n, g\ m)}, \rho_0\rangle, \langle n \rightarrow n + 5, \{\ \}\rangle)$

$= \text{Eval}\left(\langle (n,m) \rightarrow (g\ n, g\ m), \{g \rightarrow \langle n \rightarrow n+5, \emptyset\rangle\} + \rho_0 \rangle \right)$

Evaluate pair_map f

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

$\rho_1 = \{\text{pair\_map} \rightarrow <g \rightarrow \text{fun } (n,m) \text{ -> } (g\ n, g\ m), \rho_0>,$
$\quad f \rightarrow <n \rightarrow n + 5, \{\ \}>\}$

$\text{Eval}(\text{pair\_map } f, \rho_1)$

$= \text{App } (<g \rightarrow \text{fun } (n,m) \text{ -> } (g\ n, g\ m), \rho_0>, <n \rightarrow n + 5, \{\ \}>)$

$= \text{Eval } (\text{fun } (n,m)\text{->}(g\ n, g\ m), \{g \rightarrow <n \rightarrow n + 5, \{\ \}>\} + \rho_0)$

$= <(n,m) \rightarrow (g\ n, g\ m), \{g \rightarrow <n \rightarrow n + 5, \{\ \}>\} + \rho_0>$

$= <(n,m) \rightarrow (g\ n, g\ m), \{g \rightarrow <n \rightarrow n + 5, \{\ \}>,$
$\qquad f \rightarrow <n \rightarrow n + 5, \{\ \}>\}$

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 -> 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 $\to$ <(n,m) $\to$(g n, g m),
{g $\to$ <n $\to$ n + 5, { }>,
f $\to$ <n $\to$ n + 5, { }>}>,
pair_map $\to$ <g $\to$ fun (n,m) -> (g n, g m),
{f $\to$ <n $\to$ n + 5, { }>}>}

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

Evaluate f (4,6);;

$\rho_2 = \{f \to <(n,m) \to (g\ n,\ g\ m),$
$\{g \to <n \to n + 5, \{\ \}>,$
$f \to <n \to n + 5, \{\ \}>\}>,$
$pair\_map \to <g \to fun\ (n,m) \text{ -> } (g\ n,\ g\ m),$
$\{f \to <n \to n + 5, \{\ \}>\}>\}$

$Eval(f\ (4,6), \rho_2) =$
$= App(\ Eval(f, \rho_2), Eval(\ (4,6), \rho_2)\ )$
$= App(<(n,m) \to (g\ n,\ g\ m), \{g \to <n \to n + 5, \{\ \}>,$
$f \to <n \to n + 5, \{\ \}>\}>,$
$(4,6)) =$ $Eval((g\ n,\ g\ m), \{n \to 4, m \to 6,$
$g \to <n, n+5>,$
$f \to <n, n+5>\})$

Evaluate f (4,6);;

$App(<(n,m) \to (g\ n, g\ m), \{g \to <n \to n + 5, \{\ \}>,$
$f \to <n \to n + 5, \{\ \}>\}>,$
$(4,6))$

$= Eval((g\ n, g\ m), \rho_7)$

where $\rho_7 = \{n \to 4, m \to 6\} +$
$\{g \to <n \to n + 5, \{\ \}>,$
$f \to <n \to n + 5, \{\ \}>\}$

$=$

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), ρ_7)

where ρ_7 = {n → 4, m → 6} +
{g → <n → n + 5, { }>,
f → <n → n + 5, { }>}

= (Eval (g n, ρ_7), Eval (g m, ρ_7))

= (App (Eval (g, ρ_7), Eval (n, ρ_7))),
App (Eval (g, ρ_7), Eval (m, ρ_7))))

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

= …

Evaluate f (4,6);;

Eval(n, {n -> 4}) + Eval(5, {n->4})

4 + 5

(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)

A small subset of Ocaml

$$e ::= c \mid x \mid (e_1, ..., e_n) \mid op\ e \mid e_1\ e_2 \mid fun\ x \rightarrow e \mid let\ (x_1, \ldots x_n) = (e_1, \ldots, e_n)\ in\ e$$
$$\mid if\ e\ then\ e_1\ else\ e_2$$
$$\mid match\ e_0\ with\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n$$

Syntax for expressions.

- op is a primitive operator (like +, +., or ^)
- p_i is a pattern
- let x = e in e' is synonymous to let (x) = e in e'
- No recursion; no lists; no free algebraic datatypes
- No pattern matching in let statements

Values and evaluation

Values: a subset of expressions and closures

$$v, v_i ::= c \mid (v_1, \ldots, v_n) \mid \langle x \to e, \rho \rangle$$

Evaluation

Partial-functions from expressions to values.
Evaluation of an expression may be undefined if
evaluation of it does not halt, or expression is ill-formed
or calls a primitive operation that's not defined on a value, etc.

Rules of evaluation

$$
\begin{aligned}
Eval(c,\rho) &= c \\
Eval(x,\rho) &= \rho(x), \quad \textit{if } \rho(x) \textit{ is defined} \\
&= \textit{undefined, otherwise} \\
Eval((e_1,\ldots,e_n),\rho) &= (v_1,\ldots,v_n), \;\; \textit{where } Eval(e_n,\rho) = v_n, \ldots, Eval(e_1,\rho) = v_1 \\
Eval(op\ e) &= [|op|](v), \;\; \textit{where } Eval(e) = v \\
Eval(e_1\ e_2,\rho) &= App(Eval(e_1,\rho), Eval(e_2,\rho)) \\
Eval(fun\ x \rightarrow e,\rho) &= \langle x \rightarrow e, \rho\rangle \\
Eval(let\ (x_1,\ldots,x_n) = (e_1,\ldots,e_n)\ in\ e,\rho) &= Eval(e, \{x_1 \rightarrow v_1, \ldots x_n \rightarrow v_n\} + \rho) \\
&\quad \textit{where for each } i, Eval(e_i,\rho) = v_i \\
Eval(if\ e\ then\ e_1\ else\ e_2,\rho) &= Eval(e_1,\rho) \qquad \textit{if } Eval(e,\rho) = true \\
&= Eval(e_2,\rho) \qquad \textit{if } Eval(e,\rho) = false \\
Eval(match\ e\ with\ p_1 \rightarrow e_1 \mid \ldots \mid p_n \rightarrow e_n,\rho) &= Eval(e_1, b_1 + \rho) \qquad \textit{if } Eval(e,\rho) \textit{ matches } p_1 \textit{ producing binding } b_1 \\
&= \ldots \\
&= Eval(e_n, b_n + \rho) \qquad \textit{if } Eval(e,\rho) \textit{ matches } p_n \textit{ producing binding } b_n \\
&= \textit{undefined} \qquad\qquad \textit{otherwise}
\end{aligned}
$$

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 =
    (let two = (print_string "test\n"; 2) in
        (fun x -> x + two));;
test
val add_two : int -> int = <fun>

# let add2 =
    fun x -> (let two = (print_string "test\n"; 2) in (x+two));;
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

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