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 $(f \ e_1)$
  - Evaluate $e_1$ first
  - Then evaluate $f$ on the value of $e_1$
Booleans (aka Truth Values)

Order of Evaluation

# true;;
- : bool = true

# false;;
- : bool = false

// \( \rho_7 = \{c \to 4, \text{test} \to 3.7, a \to 1, b \to 5\} \)

# 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

```ocaml
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

// ρ7 = {c → 4, test → 3.7, a → 1, b → 5}
# let s = (5,"hi",3.2);;
val s : int * string * float = (5, "hi", 3.2)

// ρ8 = {s → (5, "hi", 3.2), c → 4, test → 3.7, a → 1, b → 5}
Pattern Matching with Tuples

/ \rho_8 = \{s \to (5, "hi", 3.2),
c \to 4, \text{test} \to 3.7,
a \to 1, b \to 5\}\n
# 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:

  \[
  f \rightarrow < (v_1, \ldots, v_n) \rightarrow \text{exp}, \rho_f >
  \]

- Where \( \rho_f \) 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")
# 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>
Closure for plus_pair

- Assume $\rho_{plus\_pair}$ was the environment just before plus_pair defined

- Closure for plus_pair:
  $\langle (n,m) \rightarrow n + m, \rho_{plus\_pair} \rangle$

- Environment just after plus_pair defined:
  $\{ plus\_pair \rightarrow \langle (n,m) \rightarrow n + m, \rho_{plus\_pair} \rangle \}$
  $\rho_{plus\_pair}$
# 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>
Closure for plus_pair

- Assume $\rho_{\text{plus\_pair}}$ was the environment just before plus_pair defined

- Closure for plus_pair:

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

- Environment just after plus_pair defined:

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

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

```ocaml
# 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 $\rho_{\text{plus_pair}}$ was the environment just before plus_pair defined

- Closure for plus_pair:
  $$<(n,m) \rightarrow n + m, \rho_{\text{plus_pair}}>$$

- Environment just after plus_pair defined:
  $$\{\text{plus_pair} \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus_pair}} >\} + \rho_{\text{plus_pair}}$$
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 `;';
Partial application of functions

```ocaml
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*
How exactly do we evaluate expressions to values?
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:
  \[ \rho_{\text{plus}_x} = \{ ..., x \rightarrow 12, ... \} \]

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

- Closure for fun y -> y + x:
  \[ <y \rightarrow y + x, \rho_{\text{plus}_x} > \]
  = fu body env it was created in

- Environment just after plus_x defined:
  \[ \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x} > \} + \rho_{\text{plus}_x} \]
Evaluating declarations

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

- Update: $\rho_1 + \rho_2$ has all the bindings in $\rho_1$ and all those in $\rho_2$ that are not rebound in $\rho_1$
  
  $\{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}\} + \{y \rightarrow 100, b \rightarrow 6\}$
  
  $= \{x \rightarrow 2, y \rightarrow 3, a \rightarrow \text{“hi”}, b \rightarrow 6\}$
Evaluating expressions

- Evaluation uses an environment $\rho$
- A constant evaluates to itself
- To evaluate an variable, look it up in $\rho(\nu)$
- To evaluate uses of $+,-,\ldots$, etc, eval args, then do operation
- Function expression evaluates to its closure
- To evaluate a local dec: $\text{let } x = e_1 \text{ in } e_2$
  - Eval $e_1$ to $\nu$, eval $e_2$ using $\{x \rightarrow \nu\} + \rho$
More generally, $\text{let pattern } = e_1 \text{ in } e_2$
Evaluating expressions

- To evaluate \( \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \) under an environment \( \rho \)
  
  Eval \( e_1 \) using \( \rho \)
  
  If it evaluates to true, then eval \( e_2 \) using \( \rho \).
  
  Else, (i.e., it evaluates to false), then eval \( e_3 \) using \( \rho \).

- To evaluate \( \text{match } e \text{ with } p_1 \rightarrow e_1 \mid p_2 \rightarrow e_2 \mid \ldots \mid p_n \rightarrow e_n \text{ in env } \rho \)
  
  Evaluate \( e \) to \( v \).
  
  If \( v \) matches \( p_1 \), then eval \( e_1 \), else
  
  if \( v \) matches \( p_2 \), then eval \( e_2 \), else ...
    
  if \( v \) matches \( p_n \) then eval \( e_n \)
  
  (all evaluations using environment \( \rho \))
Evaluation of Application with Closures

- Given application expression \( f(e_1,\ldots,e_n) \) in env \( \rho \), where \( f \) is an identifier.

- In environment \( \rho \), evaluate left term to closure,
  \[
  c = \langle (x_1,\ldots,x_n) \rightarrow b, \rho \rangle
  \]

- \( (x_1,\ldots,x_n) \) variables in (first) argument

- Evaluate \( (e_1,\ldots,e_n) \) to value \( (v_1,\ldots,v_n) \) using \( \rho \)

- Update the environment \( \rho \) to
  \[
  \rho' = \{ x_1 \rightarrow v_1,\ldots, x_n \rightarrow v_n \} + \rho
  \]

- Evaluate body \( b \) in environment \( \rho' \)
Evaluation of Application of plus_x;;

- Have environment:
  \[ \rho = \{ \text{plus}_x \mapsto <y \mapsto y + x, \rho_{\text{plus}_x}>, \ldots, y \mapsto 3, \ldots \} \]
  where \[ \rho_{\text{plus}_x} = \{ x \mapsto 12, \ldots, y \mapsto 24, \ldots \} \]

- Eval \((\text{plus}_x \ y, \ \rho)\) rewrites to
  \[
  \text{App} (\text{Eval}(\text{plus}_x, \ \rho), \ \text{Eval}(y, \ \rho)) \text{ rewrites to} \\
  \text{App} (\ <y \mapsto y + x, \ \rho_{\text{plus}_x} >, 3) \text{ rewrites to} \\
  \text{Eval} (y + x, \ \{y \mapsto 3\} + \rho_{\text{plus}_x}) \text{ rewrites to} \\
  \text{Eval} (3 + 12 , \ \rho_{\text{plus}_x} ) = 15
  \]
**App**

\[
\text{let add } x \; y = x + y
\]

\[
\text{let add } = \text{fun } x \to (\text{fun } y \to (x + y))
\]

**App (closure, value)**

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

**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.

\[
<y \to x + y, \{x \to 5\}>
\]

9/6/2018
Closure question

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

```
let f = fun n -> n + 5;;  (* 0 *)
(* 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

\[
\begin{align*}
\text{let } f &= \text{fun } n \rightarrow n + 5; \\
\rho_0 &= \{ f \rightarrow \langle n \rightarrow n + 5, \{} \rangle \}
\end{align*}
\]
If we start in an empty environment, and we execute:

```plaintext
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 *)?
\( \rho_0 = \{ f \rightarrow <n \rightarrow n + 5, \{ \} > \} \)

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

\( \rho_1 = \{ \text{pair_map} \rightarrow \)
\( <g \rightarrow \text{fun (n,m) ->} (g n, g m), \{ f \rightarrow <n \rightarrow n + 5, \{ \} > \} >, \)
\( f \rightarrow <n \rightarrow n + 5, \{ \} > \} \}

I.e.
\( \rho_1 = \{ \text{pair_map} \rightarrow \)
\( <g \rightarrow \text{fun (n,m) ->} (g n, g m), \rho_0 >, \)
\( f \rightarrow <n \rightarrow n + 5, \{ \} > \} \} \)
Closure question

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

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

(* 2 *)
```

```ml
let a = f (4,6);;
```

What is the environment at (* 2 *)?
Evaluate `pair_map f`

\[ \rho_0 = \{ f \mapsto <n \mapsto n + 5, \{\} > \} \]

\[ \rho_1 = \{ \text{pair_map} \mapsto <g \mapsto \text{fun (n,m) -> (g n, g m)}, \rho_0 >, \]
\[ \quad f \mapsto <n \mapsto n + 5, \{\} > \} \]

`let f = pair_map f;;`
Evaluate \texttt{pair\_map f}

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

\(\rho_1 = \{ \text{pair\_map} \rightarrow <g \rightarrow \text{fun} (n,m) \rightarrow (g\ n,\ g\ m),\ \rho_0>,\)

\(\quad f \rightarrow <n \rightarrow n + 5, \{ \} >\}\)

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

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

$\rho_1 = \{ \text{pair\_map} \mapsto <g \mapsto \text{fun} (n,m) \rightarrow (g\ n, g\ m), \rho_0 >, \}

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

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

$\text{= Eval (}
\frac{(n,m) \rightarrow (g\ n, g\ m),}{\begin{array}{l}
g \mapsto n \mapsto n + 5, \\
\rho_0 \mapsto \rho_1
\end{array}}$

9/6/2018
Evaluate \text{pair\_map } f

\rho_0 = \{f \mapsto \langle n \mapsto n + 5, \{ \} \rangle\}
\rho_1 = \{\text{pair\_map} \mapsto \langle g \mapsto \text{fun } (n, m) \rightarrow (g \ n, g \ m), \rho_0 \rangle, f \mapsto \langle n \mapsto n + 5, \{ \} \rangle\}

\text{Eval}(\text{pair\_map } f, \ \rho_1)
= \text{App } (\langle g \mapsto \text{fun } (n, m) \rightarrow (g \ n, g \ m), \rho_0 \rangle, \langle n \mapsto n + 5, \{ \} \rangle)
= \text{Eval } (\text{fun } (n, m) \rightarrow (g \ n, g \ m), \{g \mapsto \langle n \mapsto n + 5, \{ \} \rangle\} + \rho_0)
= \langle (n, m) \mapsto (g \ n, g \ m), \{g \mapsto \langle n \mapsto n + 5, \{ \} \rangle\} + \rho_0 \rangle
= \langle (n, m) \mapsto (g \ n, g \ m), \{g \mapsto \langle n \mapsto n + 5, \{ \} \rangle, f \mapsto \langle n \mapsto n + 5, \{ \} \rangle\} \rangle
\[ \rho_1 = \{ \text{pair\_map} \rightarrow \langle \text{g} \rightarrow \text{fun} (n,m) \rightarrow (g\ n,\ g\ m), \{f \rightarrow <n \rightarrow n + 5, \{\}\}\rangle \rangle, \]
\[ \quad f \rightarrow <n \rightarrow n + 5, \{\}\}\rangle \rangle \}
\]

let f = pair\_map f;;

\[ \rho_2 = \{ f \rightarrow <(n,m) \rightarrow (g\ n,\ g\ m), \]
\[ \quad \{g \rightarrow <n \rightarrow n + 5, \{\}\}\rangle, \]
\[ \quad f \rightarrow <n \rightarrow n + 5, \{\}\rangle\rangle, \]
\[ \quad \text{pair\_map} \rightarrow <\text{g} \rightarrow \text{fun} (n,m) \rightarrow (g\ n,\ g\ m), \]
\[ \quad \{f \rightarrow <n \rightarrow n + 5, \{\}\}\rangle\rangle \} \]
If we start in an empty environment, and we execute:

```plaintext
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);;
```

What is the environment at (* 3 *)?
\[ \rho_2 = \{ f \mapsto <(n,m) \mapsto (g \ n, \ g \ m), \]
\[ \quad \{ g \mapsto <n \mapsto n + 5, \{ \}>, \]
\[ \quad \ f \mapsto <n \mapsto n + 5, \{ \}>\}, \]
\[ \quad \text{pair_map} \mapsto <g \mapsto \text{fun} \ (n,m) \to (g \ n, g \ m), \]
\[ \quad \quad \{ f \mapsto <n \mapsto n + 5, \{ \}>\}> \} \]

let a = f (4,6);;
Evaluate $f(4,6)$;

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

$\text{Eval}(f(4,6), \rho_2) = \\
\text{App( \text{Eval}(f, \rho_2), \text{Eval}( (4,6), \rho_2) )}$
Evaluate $f(4,6)$;

$$\rho_2 = \{ f \rightarrow \langle n,m \rangle \rightarrow (g\ n,\ g\ m), \quad \{ g \rightarrow \langle n \rightarrow n + 5, \ \{ \ } \rangle \rangle, \quad f \rightarrow \langle n \rightarrow n + 5, \ \{ \ } \rangle \rangle, \quad \text{pair}_\text{map} \rightarrow \langle g \rightarrow \text{fun} (n,m) \rightarrow (g\ n,\ g\ m), \quad \{ f \rightarrow \langle n \rightarrow n + 5, \ \{ \ } \rangle \rangle \rangle \}$$

$$\text{Eval}(f(4,6), \rho_2) =$$

$$\text{App}(\text{Eval}(f, \rho_2), \text{Eval}((4,6), \rho_2))$$

$$\text{App}(\langle (n,m) \rightarrow (g\ n,\ g\ m), \ \{ g \rightarrow \langle n \rightarrow n + 5, \ \{ \ } \rangle \rangle, \quad f \rightarrow \langle n \rightarrow n + 5, \ \{ \ } \rangle \rangle, \quad (4,6)) = \text{Eval}(\langle (9\ n,\ 9\ m), \ \{ n \rightarrow 4,\ m \rightarrow 6 \}
\{ g \rightarrow \langle n,\ n+5 \rangle, \quad f \rightarrow \langle n,\ 4+5 \rangle \})$$
Evaluate \( f(4,6) \);

\[
\text{App}(<(n,m) \to (g \; n, g \; m), \{g \to <n \to n + 5, \{\}\}, f \to <n \to n + 5, \{\}\}>, (4,6)) = \text{Eval}((g \; n, g \; m), \rho_7)
\]

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

= \]

= \]

= \]

= \]
Evaluate \( f(4,6) \):

\[
\text{App}(<(n,m) \rightarrow (g\ n, g\ m), \{g \rightarrow <n \rightarrow n + 5, \{ \}>, f \rightarrow <n \rightarrow n + 5, \{ \}>>),
\]

\( (4,6) \)

\( = \text{Eval}((g\ n, g\ m), \rho_7) \)

where \( \rho_7 = \{n \rightarrow 4, m \rightarrow 6\} + \)

\[
\{g \rightarrow <n \rightarrow n + 5, \{ \}>, f \rightarrow <n \rightarrow n + 5, \{ \}>\}
\]

\( = (\text{Eval} (g\ n, \rho_7), \text{Eval} (g\ m, \rho_7)) \)

\( = (\text{App} (\text{Eval} (g, \rho_7), \text{Eval} (n, \rho_7))), \text{App} (\text{Eval} (g, \rho_7), \text{Eval} (m, \rho_7))) \)

\( = (\text{App}(<n \rightarrow n + 5, \{ \}>, 4), \text{App}(<n \rightarrow n + 5, \{ \}>, 6)) \)

\( = ... \)
Evaluate $f(4, 6)$;

\[
\begin{align*}
\text{App}(\langle n \rightarrow n + 5, \{ \} \rangle, 4), \\
\text{App}(\langle n \rightarrow n + 5, \{ \} \rangle, 6)) = \\
(\text{Eval}(n + 5, \{n \rightarrow 4\} + \{\}), \\
\text{Eval}(n + 5, \{n \rightarrow 6\} + \{\})) = \\
(\text{Eval}(4 + 5, \{n \rightarrow 4\} + \{\}), \\
\text{Eval}(6 + 5, \{n \rightarrow 6\} + \{\})) = (9, 11)
\end{align*}
\]
A small subset of Ocaml

\[ e ::= c \mid x \mid (e_1, \ldots, e_n) \mid op\ e \mid e_1 \; e_2 \mid fun\ x \to\ e \mid let\ (x_1, \ldots, x_n) = (e_1, \ldots, e_n)\ in\ e \]
\[ \quad \mid if\ e\ then\ e_1\ else\ e_2 \]
\[ \quad \mid match\ e_0\ with\ p_1 \to e_1 \mid \ldots \mid p_n \to e_n \]

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

\[ \text{Eval}(c, \rho) = c \]
\[ \text{Eval}(x, \rho) = \rho(x), \quad \text{if } \rho(x) \text{ is defined} \]
\[ = \text{undefined}, \text{otherwise} \]
\[ \text{Eval}(e_1, \ldots, e_n, \rho) = (v_1, \ldots, v_n), \quad \text{where } \text{Eval}(e_n, \rho) = v_n, \ldots, \text{Eval}(e_1, \rho) = v_1 \]
\[ \text{Eval}(\text{op } e) = [\text{op}(v), \quad \text{where } \text{Eval}(e) = v \]
\[ \text{Eval}(e_1 \ e_2, \rho) = \text{App}(\text{Eval}(e_1, \rho), \text{Eval}(e_2, \rho)) \]
\[ \text{Eval}(\text{fun } x \to e, \rho) = \lambda x \to e, \rho \]
\[ \text{Eval}(\text{let } (x_1, \ldots, x_n) = (e_1, \ldots, e_n) \text{ in } e, \rho) = \text{Eval}(\{x_1 \to v_1, \ldots, x_n \to v_n\}, \rho) \]
\[ \text{where for each } i, \text{Eval}(x_i, \rho) = v_i \]
\[ \text{Eval}(\text{if } e \text{ then } e_1 \text{ else } e_2, \rho) = \text{Eval}(e_1, \rho) \quad \text{if } \text{Eval}(e, \rho) = \text{true} \]
\[ = \text{Eval}(e_2, \rho) \quad \text{if } \text{Eval}(e, \rho) = \text{false} \]
\[ \text{Eval}(\text{match } e \text{ with } p_1 \to e_1 | \ldots | p_n \to e_n, \rho) = \text{Eval}(e_1, b_1 + \rho) \quad \text{if } \text{Eval}(e, \rho) \text{ matches } p_1 \text{ producing binding } b_1 \]
\[ = \ldots \]
\[ = \text{Eval}(e_n, b_n + \rho) \quad \text{if } \text{Eval}(e, \rho) \text{ matches } p_n \text{ producing binding } b_n \]
\[ = \text{undefined} \quad \text{otherwise} \]
Functions as arguments

```ocaml
# 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:

```ocaml
# 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:

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

- How do you write thrice with compose?
Thrice

- Recall:

\[
\text{let thrice } f \ x = f \ (f \ (f \ x));
\]

val thrice : (\text{'a} \rightarrow \text{a}) \rightarrow \text{a} \rightarrow \text{a} = \langle\text{fun}\rangle

- How do you write thrice with compose?

\[
\text{let thrice } f = \text{compose} \ f \ (\text{compose} \ f \ f);
\]

val thrice : (\text{a} \rightarrow \text{a}) \rightarrow \text{a} \rightarrow \text{a} = \langle\text{fun}\rangle

- Is this the only way?
Lambda lifting

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

```ocaml
# 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
test
- : int = 11

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