Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC

http://courses.engr.illinois.edu/cs421

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

Try steps 1 - 3 from WA0-practice
Functions

# let plus_two n = n + 2;;
val plus_two : int -> int = <fun>
# plus_two 17;;
- : int = 19
# let plus_two = fun n -> n + 2;;
val plus_two : int -> int = <fun>
# plus_two 14;;
- : int = 16

First definition syntactic sugar for second

9/1/16
Using a nameless function

# (fun x -> x * 3) 5;; (* An application *)
- : int = 15

# ((fun y -> y +. 2.0), (fun z -> z * 3));; (* As data *)
- : (float -> float) * (int -> int) = (<fun>, <fun>)

Note: in fun v -> exp(v), scope of variable is only the body exp(v)
Values fixed at declaration time

```ocaml
# let x = 12;;
val x : int = 12

# let plus_x y = y + x;;
val plus_x : int -> int = <fun>

# plus_x 3;;
```

What is the result?
Values fixed at declaration time

```ocaml
# let x = 12;;
val x : int = 12

# let plus_x y = y + x;;
val plus_x : int -> int = <fun>

# plus_x 3;;
- : int = 15
```
Values fixed at declaration time

# let x = 7;; (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;

What is the result this time?
Values fixed at declaration time

# let x = 7;; (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;

What is the result this time?
Values fixed at declaration time

# let x = 7;;  (* New declaration, not an update *)
val x : int = 7

# plus_x 3;;
- : int = 15
Question

- Observation: Functions are first-class values in this language

- Question: What value does the environment record for a function variable?

- Answer: a closure
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:

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

Where \( \rho_f \) is the environment in effect when \( f \) is defined (if \( f \) is a simple function)
Closure for plus_x

- When plus_x was defined, had environment:

  \[ \rho_{\text{plus}_x} = \{ \ldots, x \rightarrow 12, \ldots \} \]

- Recall: \texttt{let plus}_x y = y + x

  is really \texttt{let plus}_x = \texttt{fun y -> y + x}

- Closure for \texttt{fun y -> y + x}:

  \[ <y \rightarrow y + x, \rho_{\text{plus}_x}> \]

- Environment just after plus_x defined:

  \[ \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x}> \} + \rho_{\text{plus}_x} \]
Now it’s your turn

You should be able to do remainder of WA0-practice
```
# 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:

$$n^2 = (2 \times n - 1) + (n - 1)^2$$

```ocaml
# 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

```ocaml
# 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
Lists

- First example of a recursive datatype (aka algebraic datatype)

- Unlike tuples, lists are homogeneous in type (all elements same type)
Lists

- List can take one of two forms:
  - Empty list, written [ ]
  - Non-empty list, written \( x :: xs \)
    - \( x \) is head element, \( xs \) is tail list, :: called “cons”
  - Syntactic sugar: \([x] == x :: [ ]\)
  - \([ x1; x2; \ldots; xn ] == x1 :: x2 :: \ldots :: xn :: [ ]\)
Lists

# let fib5 = [8;5;3;2;1;1];;
val fib5 : int list = [8; 5; 3; 2; 1; 1]
# let fib6 = 13 :: fib5;;
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
# (8::5::3::2::1::1::[ ]) = fib5;;
- : bool = true
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
Lists are Homogeneous

```ocaml
# let bad_list = [1; 3.2; 7];;

Characters 19-22:
  let bad_list = [1; 3.2; 7];;;

^^^^

This expression has type float but is here used with type int
```
Question

Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[“hi”; “there”]; [“wahcha”]; [ ]; [“doin”]]
Which one of these lists is invalid?

1. [2; 3; 4; 6]
2. [2,3; 4,5; 6,7]
3. [(2.3,4); (3.2,5); (6,7.2)]
4. [[[“hi”; “there”]]; [“wahcha”]; [ ]; [“doin”]]

3 is invalid because of last pair
Functions Over Lists

# let rec double_up list =
  match list
  with [ ] -> [ ]  (* pattern before ->, expression after *)
  | (x :: xs) -> (x :: x :: double_up xs);;

val double_up : 'a list -> 'a list = <fun>
# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1]
Functions Over Lists

# let silly = double_up ["hi"; "there"];;
val silly : string list = ["hi"; "hi"; "there"; "there"]
# let rec poor_rev list =
    match list
    with [] -> []
     | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
# poor_rev silly;;
- : string list = ["there"; "there"; "hi"; "hi"]
Question: Length of list

- Problem: write code for the length of the list
  - How to start?

```plaintext
let length l =
```
Question: Length of list

- Problem: write code for the length of the list
  - How to start?

let rec length l =
    match l with
Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```ml
let rec length l =
  match l with
```
Question: Length of list

- Problem: write code for the length of the list
  - What patterns should we match against?

```plaintext
let rec length l =
    match l with [] ->
    | (a :: bs) ->
```
Question: Length of list

Problem: write code for the length of the list
  - What result do we give when \( l \) is empty?

```ocaml
def length l =
  match l with [] -> 0
  | (a :: bs) ->
```
Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when \( l \) is not empty?

```ocaml
let rec length l =
  match l with
  | [] -> 0
  | (a :: bs) ->
```
Question: Length of list

- Problem: write code for the length of the list
  - What result do we give when \( l \) is not empty?

\[
\text{let rec length } l = \\
\text{match } l \text{ with } [\text] -> 0 \\
| (a :: bs) -> 1 + \text{length } bs
\]
How can we efficiently answer if two lists have the same length?
How can we efficiently answer if two lists have the same length?

```ocaml
let rec same_length list1 list2 =
    match list1 with [] ->
        (match list2 with [] -> true
        | (y::ys) -> false)
    | (x::xs) ->
        (match list2 with [] -> false
        | (y::ys) -> same_length xs ys)
```

9/1/16
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
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

```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;;
val g : int -> int = <fun>
- : int = 10

# thrice (fun s -> "Hi! " ^ s) "Good-bye!";;
val thrice : ('a -> 'a) -> 'a -> 'a = <fun>
- : 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")
```
Closure for plus_pair

- Assume $\rho_{\text{plus}_\text{pair}}$ was the environment just before plus_pair defined.
- Closure for plus_pair:
  
  $$<(n,m) \rightarrow n + m, \rho_{\text{plus}_\text{pair}}>$$

- Environment just after plus_pair defined:

  $$\{\text{plus}_\text{pair} \rightarrow <(n,m) \rightarrow n + m, \rho_{\text{plus}_\text{pair}} > \} + \rho_{\text{plus}_\text{pair}}$$
Consider this code:

```ocaml
define x = 27

let f x = 
  let x = 5 in
  (fun x -> print_int x) 10
f 12
```

What value is printed?

- 5
- 10
- 12
- 27
Recall: let plus_x = fun x => y + x

let x = 12
X \rightarrow 12

... 

let plus_x = fun y => y + x

... 

plus_x \rightarrow 

y \rightarrow y + x 

x \rightarrow 12

let x = 7

... 

x \rightarrow 7
Closure for plus\_x

- When plus\_x was defined, had environment:

\[ \rho_{\text{plus}_x} = \{..., x \to 12, ...\} \]

- Recall: \texttt{let plus\_x y = y + x}
  is really \texttt{let plus\_x = fun y -> y + x}

- Closure for \texttt{fun y -> y + x}:

  \[ <y \to y + x, \rho_{\text{plus}_x}> \]

- Environment just after plus\_x defined:

\[ \{\text{plus\_x} \to <y \to y + x, \rho_{\text{plus}_x}>\} + \rho_{\text{plus}_x} \]
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")
```
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:

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

Where \(\rho\) is the environment in effect when the function is defined (for a simple function)
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:
  \[
  \{\text{plus\_pair} \rightarrow (n,m) \rightarrow n + m, \rho_{\text{plus\_pair}}\} + \rho_{\text{plus\_pair}}
  \]
Functions with more than one argument

```ocaml
# 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

```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
```
Your turn now

Try later parts from WA1

Caution!

Know what the argument is and what the body is
Functions as arguments

```ml
# 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!"
```
Evaluating declarations

- Evaluation uses an environment $\rho$
- To evaluate a (simple) declaration `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 $\rho_1$

$\{x \rightarrow 2, y \rightarrow 3, a \rightarrow "hi"\} + \{y \rightarrow 100, b \rightarrow 6\} = \{x \rightarrow 2, y \rightarrow 3, a \rightarrow "hi", b \rightarrow 6\}$
Evaluating expressions

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

- 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 the right term to value $(v_1,\ldots,v_n)$
- 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:
  \[ \rho = \{ \text{plus}_x \rightarrow \langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, \ldots, y \rightarrow 3, \ldots \} \]

  where \( \rho_{\text{plus}_x} = \{ x \rightarrow 12, \ldots \} \)

- \( \text{Eval} (\text{plus}_x \ y, \ \rho) \) rewrites to

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

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

- \( \text{Eval} (3 + 12, \rho_{\text{plus}_x}) = 15 \)
Evaluation of Application of plus_pair

- Assume environment

\[ \rho = \{ x \to 3, \ldots, \text{plus_pair} \to <(n,m) \to n + m, \rho_{\text{plus_pair}} > \} + \rho_{\text{plus_pair}} \]

- Eval (plus_pair (4,x), \rho) =

- App (<(n,m) \to n + m, \rho_{\text{plus_pair}} >, (4,3)) =

- Eval (n + m, \{ n \to 4, m \to 3 \} + \rho_{\text{plus_pair}} ) =

- Eval (4 + 3, \{ n \to 4, m \to 3 \} + \rho_{\text{plus_pair}} ) = 7
If we start in an empty environment, and we execute:

```ocaml
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;;

\( \rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \)
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);;
(* 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);;

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

- If we start in an empty environment, and we execute:
  
  ```ml
  let f = fun n => 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

\[ \rho_0 = \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \]
\[ \rho_1 = \{ \text{pair_map} \rightarrow \langle g \rightarrow \text{fun (n,m) -> (g n, g m)}, \rho_0 \rangle, \]
\[ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \} \]

let f = pair_map f;;
Evaluate pair_map f

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

\[ \rho_1 = \{ \text{pair_map} \rightarrow <g \rightarrow \text{fun (n,m) -> (g n, g m)}, \rho_0 \}, \]

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

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

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

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

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

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

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

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

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

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

$$<n \rightarrow n + 5, \{ \}\}\}) =$$

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

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

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

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

let f = pair_map f;;

\( \rho_2 = \{ f \rightarrow < (n,m) \rightarrow (g n, g m), \\
\{ g \rightarrow < n \rightarrow n + 5, \{ \} > > \}, \\
f \rightarrow < n \rightarrow n + 5, \{ \} > > \} \)

pair_map \rightarrow < g \rightarrow \text{fun} (n,m) -> (g n, g m), \\
\{ 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;;
let a = f (4,6);;
(* 3 *)
```

What is the environment at (* 3 *)?
\[ \rho_2 = \{ f \rightarrow \langle n,m \rangle \rightarrow (g\ n,\ g\ m), \]
\[ \{ g \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \}, \]
\[ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \}, \]
\[ \text{pair\_map} \rightarrow \langle g \rightarrow \text{fun} (n,m) \rightarrow (g\ n,\ g\ m), \]
\[ \{ f \rightarrow \langle n \rightarrow n + 5, \{ \} \rangle \}, \]}

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

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

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

$$\rho_2 = \{ f \rightarrow <(n,m) \rightarrow (g\ n,\ g\ m),$$

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

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

$$\text{pair\_map} \rightarrow <g \rightarrow \text{fun} (n,m) \rightarrow (g\ n,\ g\ m),$$

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

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

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

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

$$\text{(4,6)}) =$$
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), \{n \rightarrow 4, m \rightarrow 6\} + \\
\{g \rightarrow <n \rightarrow n + 5, \{ \}>,, f \rightarrow <n \rightarrow n + 5, \{ \}>}) = \\
\text{Eval}((\text{App}(<n \rightarrow n + 5, \{ \}>), 4), \\
\text{App}(<n \rightarrow n + 5, \{ \}>), 6)), \\
\{n \rightarrow 4, m \rightarrow 6, g \rightarrow <n \rightarrow n + 5, \{ \}>,, f \rightarrow <n \rightarrow n + 5, \{ \}>}) = \\
9/1/16
Evaluate $f(4,6);$

$$\rho_3 = \{ n \rightarrow 4, m \rightarrow 6, g \rightarrow <n \rightarrow n + 5, \{ \}>,
\hspace{1cm} f \rightarrow <n \rightarrow n + 5, \{ \}>) \}$$

Eval((App(<n \rightarrow n + 5, \{ \}>, 4),
\hspace{1cm} App (<n \rightarrow n + 5, \{ \}>, 6)), \rho_3) =

Eval((Eval(n + 5, \{n \rightarrow 4\} + \{\})),
\hspace{1cm} (Eval(n + 5, \{n \rightarrow 6\} + \{\})), \rho_3) =

Eval((Eval(4 + 5, \{n \rightarrow 4\} + \{\})),
\hspace{1cm} (Eval(6 + 5, \{n \rightarrow 6\} + \{\})), \rho_3) =

Eval((9, 11), \rho_3) = (9, 11)
Higher Order Functions

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

```ml
# 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?
Recall:

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

How do you write `thrice` with `compose`?

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

Is this the only way?
Partial Application

# (+) ; ;
- : int -> int -> int = <fun>
# (+) 2 3 ; ;
- : int = 5
# let plus_two = (+) 2 ; ;
val plus_two : int -> int = <fun>
# plus_two 7 ; ;
- : int = 9

Partial application also called sectioning
Lambda Lifting

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

```ocaml
# let add_two = (+) (print_string "test\n"; 2);;
```
```
test
```
```
val add_two : int -> int = <fun>
```

```ocaml
# 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
test
-  : int = 11

- Lambda lifting delayed the evaluation of the argument to (+) until the second argument was supplied
Partial Application and “Unknown Types”

- Recall `compose plus_two`:

```ocaml
# let f1 = compose plus_two;;
val f1 : ('_a -> int) -> '_a -> int = <fun>
```

- Compare to lambda lifted version:

```ocaml
# let f2 = fun g -> compose plus_two g;;
val f2 : ('a -> int) -> 'a -> int = <fun>
```

- What is the difference?
Partial Application and “Unknown Types”

- `'_a can only be instantiated once for an expression

```ocaml
# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
```

Characters 3-14:

```ocaml
  f1 List.length;;
```

This expression has type 'a list -> int but is here used with type int -> int
Partial Application and “Unknown Types”

- ‘a can be repeatedly instantiated

```ocaml
# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : '_a list -> int = <fun>
```
# let rec map f list =

match list
with [] -> []
| (h::t) -> (f h) :: (map f t);

val map : ('a -> 'b) -> 'a list -> 'b list = <fun>

# map plus_two fib5;;
- : int list = [10; 7; 5; 4; 3; 3]

# map (fun x -> x - 1) fib6;;
: int list = [12; 7; 4; 2; 1; 0; 0]
Iterating over lists

```ocaml
# let rec fold_left f a list =
  match list
  with [] -> a
  | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left
  (fun () -> print_string)
  ()
  ["hi"; "there"];;
hithere- : unit = ()
```
Iterating over lists

# let rec fold_right f list b =
match list with [] -> b
| (x :: xs) -> f x (fold_right f xs b);
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>

# fold_right
  (fun s -> fun () -> print_string s)
  ['"hi"'; '"there"']
  ();
therehi- : unit = ()
Structural Recursion

- Functions on recursive datatypes (e.g., lists) tend to be recursive.
- Recursion over recursive datatypes generally by structural recursion:
  - Recursive calls made to components of structure of the same recursive type.
  - Base cases of recursive types stop the recursion of the function.
Structural Recursion: List Example

```ml
# let rec length list = match list
  with [ ] -> 0 (* Nil case *)
  | x :: xs -> 1 + length xs;; (* Cons case *)

val length : 'a list -> int = <fun>
```

```ml
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [ ] is base case
- Cons case recurses on component list xs
Forward Recursion

- In Structural Recursion, split input into components and (eventually) recurse
- Forward Recursion form of Structural Recursion
- In forward recursion, first call the function recursively on all recursive components, and then build final result from partial results
- Wait until whole structure has been traversed to start building answer
Forward Recursion: Examples

```ocaml
# let rec double_up list =
    match list
    with [] -> []
    | (x :: xs) -> (x :: x :: double_up xs);;
val double_up : 'a list -> 'a list = <fun>

# let rec poor_rev list =
    match list
    with [] -> []
    | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```
Encoding Recursion with Fold

```ocaml
# let rec append list1 list2 = match list1 with
  [ ] -> list2 | x::xs -> x :: append xs list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Base Case

```
# let append list1 list2 =
  fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

Operation

```
# append [1;2;3] [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

Recursive Call
One common form of structural recursion applies a function to each element in the structure.

```ocaml
# let rec doubleList list = match list
  with [ ] -> [ ]
  | x::xs -> 2 * x :: doubleList xs;;
val doubleList : int list -> int list = <fun>
# doubleList [2;3;4];;
- : int list = [4; 6; 8]
```
Can use the higher-order recursive map function instead of direct recursion

```ocaml
# let doubleList list = 
    List.map (fun x -> 2 * x) list;;
val doubleList : int list -> int list = <fun>
#
doubleList [2;3;4];;
- : int list = [4; 6; 8]
```

- Same function, but no rec
Folding Recursion

Another common form “folds” an operation over the elements of the structure

```ocaml
# let rec multList list = match list
  with [ ] -> 1
  | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>

# multList [2;4;6];;
- : int = 48
```

Computes \((2 * (4 * (6 * 1)))\)
Folding Recursion

- multList folds to the right
- Same as:

```ocaml
# let multList list =
    List.fold_right
    (fun x -> fun p -> x * p)
    list 1;;
val multList : int list -> int = <fun>
```

```ocaml
# multList [2;4;6];;
- : int = 48
```
How long will it take?

- Remember the big-O notation from CS 225 and CS 273
- Question: given input of size $n$, how long to generate output?
- Express output time in terms of input size, omit constants and take biggest power
How long will it take?

Common big-O times:

- **Constant time** $O(1)$
  - input size doesn’t matter
- **Linear time** $O(n)$
  - double input $\Rightarrow$ double time
- **Quadratic time** $O(n^2)$
  - double input $\Rightarrow$ quadruple time
- **Exponential time** $O(2^n)$
  - increment input $\Rightarrow$ double time
Linear Time

- Expect most list operations to take linear time $O(n)$
- Each step of the recursion can be done in constant time
- Each step makes only one recursive call
- List example: `multList, append`
- Integer example: `factorial`
Quadratic Time

- Each step of the recursion takes time proportional to input
- Each step of the recursion makes only one recursive call.
- List example:

```ocaml
# let rec poor_rev list = match list
  with [] -> []
       | (x::xs) -> poor_rev xs @ [x];;
val poor_rev : 'a list -> 'a list = <fun>
```
Exponential running time

- Hideous running times on input of any size
- Each step of recursion takes constant time
- Each recursion makes two recursive calls
- Easy to write naïve code that is exponential for functions that can be linear
Exponential running time

# let rec naiveFib n = match n
  with 0 -> 0
  | 1 -> 1
  | _ -> naiveFib (n-1) + naiveFib (n-2);
val naiveFib : int -> int = <fun>
An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.
- What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a tail call)?
An Important Optimization

- When a function call is made, the return address needs to be saved to the stack so we know to where to return when the call is finished.
- What if $f$ calls $g$ and $g$ calls $h$, but calling $h$ is the last thing $g$ does (a tail call)?
- Then $h$ can return directly to $f$ instead of $g$.
Tail Recursion

- A recursive program is tail recursive if all recursive calls are tail calls.
- Tail recursive programs may be optimized to be implemented as loops, thus removing the function call overhead for the recursive calls.
- Tail recursion generally requires extra “accumulator” arguments to pass partial results.
  - May require an auxiliary function.
Tail Recursion - Example

```ocaml
# let rec rev_aux list revlist =
  match list with [ ] -> revlist
 | x :: xs -> rev_aux xs (x::revlist);
val rev_aux : 'a list -> 'a list -> 'a list = <fun>

# let rev list = rev_aux list [ ];
val rev : 'a list -> 'a list = <fun>
```

- What is its running time?
Comparison

- `poor_rev [1,2,3] =`  
  `(poor_rev [2,3]) @ [1] =`  
  `(((poor_rev [3]) @ [2]) @ [1] =`  
  `(((poor_rev [ ]) @ [3]) @ [2]) @ [1] =`  
  `([ ] @ [3]) @ [2]) @ [1] =`  
  `([3] @ [2]) @ [1] =`  
  `(3:: ([ ] @ [2])) @ [1] =`  
  `[3,2] @ [1] =`  
  `3 :: ([2] @ [1]) =`  
  `3 :: (2:: ([ ] @ [1])) = [3, 2, 1]`
Comparison

- \text{rev} \ [1,2,3] = \\
- \text{rev\_aux} \ [1,2,3] \ [\ ] = \\
- \text{rev\_aux} \ [2,3] \ [1] = \\
- \text{rev\_aux} \ [3] \ [2,1] = \\
- \text{rev\_aux} \ [\ ] \ [3,2,1] = [3,2,1]
Folding Functions over Lists

How are the following functions similar?

```ocaml
# let rec sumlist list = match list with
  [ ] -> 0 | x::xs -> x + sumlist xs;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let rec prodlist list = match list with
  [ ] -> 1 | x::xs -> x * prodlist xs;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```
Folding

# let rec fold_left f a list = match list
  with [] -> a | (x :: xs) -> fold_left f (f a x) xs;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a =
<fun>
fold_left f a [x₁; x₂;...;xₙ] = f(...(f (f a x₁) x₂)...xₙ)

# let rec fold_right f list b = match list
  with [] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
<fun>
fold_right f [x₁; x₂;...;xₙ] b = f x₁(f x₂(...(f xₙ b)...))
Folding - Forward Recursion

```ocaml
# let sumlist list = fold_right (+) list 0;;
val sumlist : int list -> int = <fun>
# sumlist [2;3;4];;
- : int = 9
# let prodlist list = fold_right ( * ) list 1;;
val prodlist : int list -> int = <fun>
# prodlist [2;3;4];;
- : int = 24
```
Folding - Tail Recursion

- # let rev list =
-   fold_left
-   (fun l -> fun x -> x :: l)  //comb op
  []  //accumulator cell
  list
Folding

- Can replace recursion by fold_right in any forward primitive recursive definition
  - Primitive recursive means it only recurses on immediate subcomponents of recursive data structure
- Can replace recursion by fold_left in any tail primitive recursive definition