Consider this code:

```ocaml
let 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

Closure for plus_x:

- When plus_x was defined, had environment:
  \( \rho_{\text{plus}_x} = \{ \ldots, x \rightarrow 12, \ldots \} \)
- Recall: let plus_x y = y + x is really let plus_x = fun y -> y + x
- Closure for fun y -> y + x:
  \( <y \rightarrow y + x, \rho_{\text{plus}_x}> \)
- Environment just after plus_x defined:
  \( \{ \text{plus}_x \rightarrow <y \rightarrow y + x, \rho_{\text{plus}_x}>, \rho_{\text{plus}_x} \} \)

Your turn now

Try Problem 1 on MP2
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:
  \(< (v_1, \ldots, v_n) \rightarrow \text{exp}, \rho >\)
- 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 \(\text{plus_pair}\) defined
- Closure for \(\text{fun (n,m) -> n + m}\):
  \(< (n,m) \rightarrow n + m, \rho_{\text{plus_pair}} >\)
- Environment just after \(\text{plus_pair}\) defined:
  \(\{ \text{plus_pair} \rightarrow < (n,m) \rightarrow n + m, \rho_{\text{plus_pair}} > \}\) + \(\rho_{\text{plus_pair}}\)

---

Your turn now

Try (** 1 **) from HW2

---

Functions with more than one argument

```ocaml
define add_three x y z = x + y + z

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

```ocaml
define add_triple (u, v, w) = u + v + w

val add_triple : int * int * int -> int = <fun>
```

Again, first syntactic sugar for second

---

Your turn now

Try Problem 2 on MP2

---

Curried vs Uncurried

- Recall
  \(\text{val add_three : int -> int -> int -> int = <fun>}\)
- How does it differ from
  ```ocaml
define add_triple (u, v, w) = u + v + w

val add_triple : int * int * int -> int = <fun>
```
- \(\text{add_three}\) is **curried**;
- \(\text{add_triple}\) is **uncurried**
Curried vs Uncurried

```ocaml
# 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
```

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!"
```

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

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

```plaintext
{x → 2, y → 3, a → "hi"} + {y → 100, b → 6} = 
{x → 2, y → 3, a → "hi"}, b → 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: let x = e1 in e2
  - Eval e1 to v, eval 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 values, $(v_1,\ldots,v_n)$
- 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 $\rho_{\text{plus}_x} = \{x \rightarrow 12, \ldots\}$
- Eval $(\text{plus}_x\ y, \rho)$ rewrites to
- App $(\langle y \rightarrow y + x, \rho_{\text{plus}_x} \rangle, 3)$ rewrites to
- Eval $(y + x, \{y \rightarrow 3\} + \rho_{\text{plus}_x})$ rewrites to
- Eval $(3 + 12, \rho_{\text{plus}_x}) = 15$

Evaluation of Application of plus_pair

- Assume environment
  - $\rho = \{x \rightarrow 3,\ldots,
    \text{plus}_\text{pair} \rightarrow \langle(n,m) \rightarrow n + m, \rho_{\text{plus}_\text{pair}}\rangle + \rho_{\text{plus}_\text{pair}}\}$
- Eval $(\text{plus}_\text{pair}(4,x), \rho) =$
- App $(\langle(n,m) \rightarrow n + m, \rho_{\text{plus}_\text{pair}}\rangle, (4,3)) =$
- Eval $(n + m, \{n \rightarrow 4, m \rightarrow 3\} + \rho_{\text{plus}_\text{pair}}) =$
- Eval $(4 + 3, \{n \rightarrow 4, m \rightarrow 3\} + \rho_{\text{plus}_\text{pair}}) = 7$

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 *)?

Your turn now

Try (* 3 *) from HW2
Closure question

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

\[
\text{let } f = \text{fun } n \rightarrow n + 5;;
\]

\[
\text{let pair_map } g (n,m) = (g n, g m);;
\]

\[
\text{(1)}
\]

\[
\text{let } f = \text{pair_map } f;;
\]

\[
\text{let } a = f (4,6);;
\]

What is the environment at (*1*)?

---

Closure question

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

\[
\text{let } f = \text{fun } n \rightarrow n + 5;;
\]

\[
\text{let pair_map } g (n,m) = (g n, g m);;
\]

\[
\text{(2)}
\]

\[
\text{let } f = \text{pair_map } f;;
\]

\[
\text{let } a = f (4,6);;
\]

What is the environment at (*2*)?

---

Evaluate \(\text{pair_map } f\)

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

\[
\rho_1 = \{ \text{pair_map} \rightarrow \}
\]

\[
<g \rightarrow \text{fun } (n,m) \rightarrow (g n, g m),
\]

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

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

\[
\text{let } f = \text{pair_map } f;;
\]

---

Evaluate \(\text{pair_map } f\)

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

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

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

\[
\text{let } f = \text{pair_map } f;;
\]

\[
\text{Eval}(\text{pair_map } f, \rho_1) =
\]
Evaluate pair_map f

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

\[ \text{Eval}(\text{pair_map} f, \rho_1) = \]

\[ \text{ Eval(app (<g \to \text{fun}(n,m) -> (g n, g m), \rho_0>, \langle n \to n + 5, \{ \rangle \}), \rho_1) = } \]

\[ \text{let } f = \text{pair_map} f; \]
\[ \rho_2 = \{ f \to \langle (n,m) -> (g n, g m), \{ g \to \langle n \to n + 5, \{ \rangle \}>, f \to \langle n \to n + 5, \{ \rangle \} \rangle >, \text{pair_map} \to <\text{g} \to \text{fun}(n,m) -> (g n, g m), \{ f \to \langle n \to n + 5, \{ \rangle \} >} \} \]

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);;

What is the environment at (* 3 *)?

Final Evaluation?

\[ \rho_2 = \{ f \to \langle (n,m) -> (g n, g m), \{ g \to \langle n \to n + 5, \{ \rangle \}>, f \to \langle n \to n + 5, \{ \rangle \} \rangle >, \text{pair_map} \to <\text{g} \to \text{fun}(n,m) -> (g n, g m), \{ f \to \langle n \to n + 5, \{ \rangle \} >} \} \]

let a = f (4,6);;

Evaluate f (4,6);;

\[ \rho_2 = \{ f \to \langle (n,m) -> (g n, g m), \{ g \to \langle n \to n + 5, \{ \rangle \}>, f \to \langle n \to n + 5, \{ \rangle \} \rangle >, \text{pair_map} \to <\text{g} \to \text{fun}(n,m) -> (g n, g m), \{ f \to \langle n \to n + 5, \{ \rangle \} >} \} \]

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

\[ \rho_2 = \{ f \rightarrow (n,m) \mapsto (g_n, g_m), \]
\[ g \mapsto n \mapsto n + 5, \}
\[ f \mapsto n \mapsto n + 5, \} \} \]
\[ \text{pair}_{\text{map}} \rightarrow \{ g \mapsto \text{fun}(n,m) \mapsto (g_n, g_m), \}
\[ g \mapsto n \mapsto n + 5, \}
\[ f \mapsto n \mapsto n + 5, \} \} \}
\]
\[ \text{Eval}(f(4,6), \rho_2) = \]
\[ \text{Eval}(\text{app}(\{(n,m) \mapsto (g_n, g_m), \}
\[ g \rightarrow n \mapsto n + 5, \}
\[ f \rightarrow n \mapsto n + 5, \} \}, (4,6)), \rho_2) = \]
\[ 9/4/14 37 \]

Your turn now

Try (* 4 *) from HW2

\begin{itemize}
\item Each clause: pattern on left, expression on right
\item Each \( x, y \) has scope of only its clause
\item Use first matching clause
\end{itemize}

Match Expressions

\# let triple_to_pair triple =
\begin{verbatim}
match triple
  with
    (0, x, y) -> (x, y)
  | (x, 0, y) -> (x, y)
  | (x, y, _) -> (x, y);
\end{verbatim}
\begin{verbatim}
val triple_to_pair : int * int * int -> int * int = <fun>
\end{verbatim}

Recursive Functions

\# let rec factorial n =
\begin{verbatim}
if n = 0 then 1 else n * factorial(n - 1);
\end{verbatim}
\begin{verbatim}
val factorial : int -> int = <fun>
\end{verbatim}
\begin{verbatim}
# factorial 5;;
\end{verbatim}
\begin{verbatim}
- : int = 120
\end{verbatim}
\begin{verbatim}
# (* rec is needed for recursive function declarations *)
\end{verbatim}
Your turn now

Try Problem 4 on MP2

Recursion Example

Compute $n^2$ recursively using:

\[ n^2 = (2 \times n - 1) + (n - 1)^2 \]

```
# 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

Lists

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

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

```
# 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::[ ]) = fib5;;
val _ = true
```

```
# fib5 @ fib6;;
- : int list = [8; 5; 3; 2; 1; 1; 13; 8; 5; 3; 2; 1; 1]
```

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; ...; xn] == x1 :: x2 :: ... :: xn :: [ ]
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"]

Answer

3 is invalid because of last pair

Functions Over Lists

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

```ocaml
# let fib5_2 = double_up fib5;;
val fib5_2 : int list = [8; 8; 5; 5; 3; 3; 2; 2; 1; 1; 1]
```

Question: Length of list

Problem: write code for the length of the list

How to start?

```ocaml
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

What patterns should we match against?

let rec length l =
    match l with [] -> 0
    | (a :: bs) -> 1 + length bs
Your turn now

Try Problem 6 on MP2

---

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 
               | _ -> false) 
      | _ -> (match list2 with 
               | [] -> false 
               | _ -> same_length xs ys)
```

---

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?

```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

# 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>

Partial Application and "Unknown Types"

Partial application and "Unknown Types"

't_a can only be instantiated once for an expression

# f1 plus_two;;
- : int -> int = <fun>
# f1 List.length;;
Characters 3-14:
  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

# f2 plus_two;;
- : int -> int = <fun>
# f2 List.length;;
- : 'a list -> int = <fun>
### Functions Over Lists

```ocaml
let rec map f list =  
  match list  
  with [] -> []  
  | (h::t) -> (f h) :: (map f t);;
```

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

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

```ocaml
# 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;;
```

```ocaml
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```ocaml
# fold_left  
  (fun () -> print_string)  
  ()  
  ["hi"; "there"];;  
therehi- : unit = ()
```

### Structural Recursion

- Functions on recursive datatypes (eg 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

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

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

```ocaml
# 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>
```

```ocaml
# 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>
```

```ocaml
Base Case          Operation          Recursive Call
```

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

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

### Mapping Recursion

- 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>
```

```ocaml
# doubleList [2;3;4];;  
- : int list = [4; 6; 8]
```

### Mapping Recursion

- 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>
```

```ocaml
# doubleList [2;3;4];;  
- : int list = [4; 6; 8]
```

### 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>
```

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

- Computes \((2 \times (4 \times (6 \times 1)))\)
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

Common big-O times:

- Constant time $O(1)$
  - input size doesn’t matter
- Linear time $O(n)$
  - double input ⇒ double time
- Quadratic time $O(n^2)$
  - double input ⇒ quadruple time
- Exponential time $O(2^n)$
  - increment input ⇒ 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:

  ```ml
  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 naive code that is exponential for functions that can be linear

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 naive code that is exponential for functions that can be linear

```ml
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)?

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

Comparison

- \( \text{poor}_\text{rev} \{1,2,3\} = \)
- \( \{\text{poor}_\text{rev} \{2,3\}\} \odot [1] = \)
- \( \{\text{poor}_\text{rev} \{3\}\} \odot [2] \odot [1] = \)
- \( \{\text{poor}_\text{rev} \{\} \odot [3]\} \odot [2] \odot [1] = \)
- \( \{3\} \odot [2] \odot [1] = \)
- \( \{3\} \odot ([2] \odot [1]) = \)
- \( \{3\} \odot ([2] \odot [1]) = [3,2,1] \)

Tail Recursion - Example

- \( \text{rev} \{1,2,3\} = \)
- \( \{\text{rev}_\text{aux} \{1,2,3\}\} \odot [1] = \)
- \( \{\text{rev}_\text{aux} \{2,3\}\} \odot [1] = \)
- \( \{\text{rev}_\text{aux} \{3\}\} \odot [1] = \)
- \( \{\} \odot [3,2,1] = [3,2,1] \)
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 - 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

```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 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

- 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

9/4/14