Example 1

```ml
let x = 1;;
let f y = x + y;;
let x = 2;;
let y = 3;;
f (x + y);;
```

Example 2

```ml
let x = 1;;
let f y = let x = 2 in fun z -> x + y + z;;
let x = 100;;
let g = (f 4);;
let y = 100;;
(g 1);;
```

Example 3

```ml
let f g =
  let x = 0 in
g 2
;;
let x = 100;;
let h y = x + y;;
f h;;
```

Static/Lexical Scoping

- For each occurrence of a variable,
  - Unique place in program text where variable defined
  - Most recent binding in environment

- Static/Lexical: Determined from the program text
  - Without executing the program

- Very useful for readability, debugging:
  - Don’t have to figure out “where” a variable got assigned
  - Unique, statically known definition for each occurrence
Alternative: dynamic scoping

let x = 100
let f y = x + y
let g x = f 0
let z = g 0
(* value of z? *)

Next

• More on recursion
• Higher-order functions
  - taking and returning functions

Factorial

let rec fact n =

Factorial

let rec fact n =

How does it execute?

let rec fact n =
  if n<=0
  then 1
  else n * fact (n-1);;
fac 3;;

How does it execute?

let rec fact n =
  if n<=0
  then 1
  else n * fact (n-1);;
fac 3;;
Tail recursion

• Tail recursion:
  - recursion where all recursive calls are followed by a return
  - in other words: not allowed to do anything between recursive call and return

Tail recursive factorial

let fact x =
  let rec helper x curr =
    if x <= 0
    then curr
    else helper (x - 1) (x * curr)
  in
  helper x 1;;

How does it execute?

let fact x =
  let rec helper x curr =
    if x <= 0
    then curr
    else helper (x - 1) (x * curr)
  in
  helper x 1;;
fact 3;;

Tail recursion

• Tail recursion:
  - recursion where all recursive calls are followed by a return
  - in other words: not allowed to do anything between recursive call and return

• Why do we care about tail recursion?
  - it turns out that tail recursion can be optimized into a simple loop
Compiler can optimize!

let fact x =
  let rec helper x curr =
    if x <= 0
      then curr
    else helper (x - 1) (x * curr)
  in
  helper x 1;;

Tail recursion summary

- Tail recursive calls can be optimized as a jump
- Part of the language specification of some languages (ie: you can count on the compiler to optimize tail recursive calls)

max function

let max x y = if x < y then y else x;;
(* return max element of list l *)
let list_max l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (max h curr) t
  in
  helper 0 l;;

concat function

(* concatenate all strings in a list *)
let concat l =
  let rec helper curr l =
    match l with
    | [] -> curr
    | h::t -> helper (curr ^ h) t
  in
  helper "" l;;
What’s the pattern?

```ml
let list_max l =  
  let rec helper curr l =  
    match l with  
      | [] -> curr  
      | h::t -> helper (max h curr) t  
    in helper 0 l;;

let concat l =  
  let rec helper curr l =  
    match l with  
      | [] -> curr  
      | h::t -> helper (curr ^ h) t  
    in helper "" l;;
```

fold, the general helper func!

```ml
(* to help us see the pattern: *)
let list_max l =  
  let rec helper curr l =  
    match l with  
      | [] -> curr  
      | h::t -> helper (max h curr) t  
    in helper 0 l;;

(* fold, the coolest function there is! *)
let rec fold f curr l =  
  match l with  
    | [] -> curr  
    | h::t -> fold f (f h curr) t;;
```

Examples of fold

```ml
let list_max =
  let rec fold f curr l =  
    match l with  
      | [] -> curr  
      | h::t -> fold f (f h curr) t;;

let concat =
  let list_max = fold max 0;;

let multiplier =
  let list_max = fold max 0;;
```

Examples of fold

```ml
let list_max =
  let rec fold f curr l =  
    match l with  
      | [] -> curr  
      | h::t -> fold f (f h curr) t;;

let concat =
  let list_max = fold max 0;;

let multiplier =
  let list_max = fold max 0;;
```
let fact n =
    multiplier (interval 1 n);;

Notice how all the recursion is
buried inside two functions:
interval and fold!

let cons x y = x::y;;
let f = fold cons [];;
(* same as:
    let f l = fold cons [] l *)