

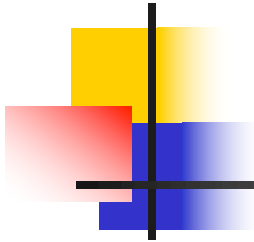
Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC



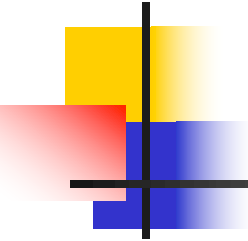
<https://courses.engr.illinois.edu/cs421/fa2017/CS421D>

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



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

```
# let fib5 = [8;5;3;2;1;1];;
```

```
val fib5 : int list = [8; 5; 3; 2; 1; 1]
```

```
# let fib6 = 13 :: fib5;;
```

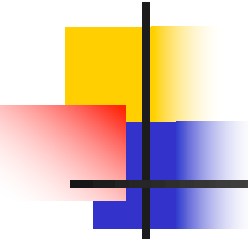
```
val fib6 : int list = [13; 8; 5; 3; 2; 1; 1]
```

```
# (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

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

Characters 19-22:

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

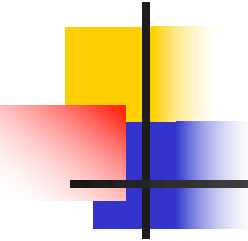
^^^

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



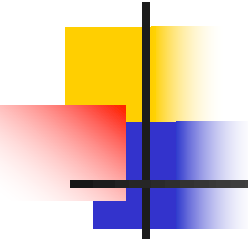
Question

- 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”]]



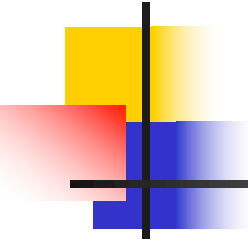
Answer

- 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



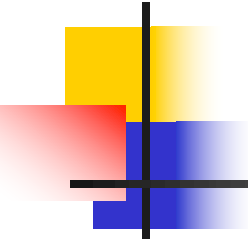
Functions Over Lists

```
# 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; 1]
```



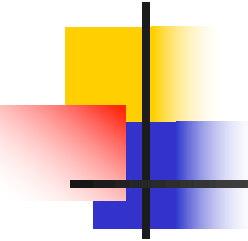
Functions Over Lists

```
# 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"]
```



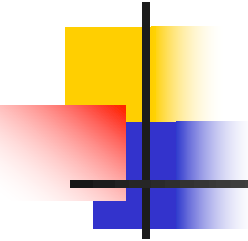
Structural Recursion

- 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



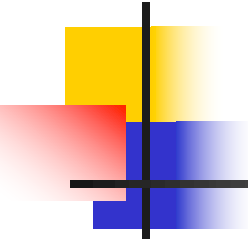
Functions Over Lists

```
# 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; 1]
```



Functions Over Lists

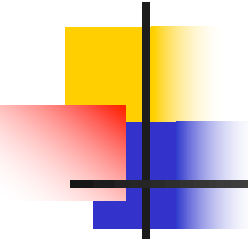
```
# let silly = double_up ["hi"; "there"];;  
val silly : string list = ["hi"; "hi"; "there"; "there"]  
# let rec poor_rev list =  
  match list  
  with [] -> []  
       | (x::xs) -> poor_rev xs @ [x];;  
val poor_rev : 'a list -> 'a list = <fun>  
# poor_rev silly;;  
- : string list = ["there";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



Your turn: `doubleList : int list -> int list`

- Write a function that takes a list of `int` and returns a list of the same length, where each element has been multiplied by 2

```
let rec doubleList list =  
  match list  
  with [] -> []  
       | x :: xs -> (2 * x) :: doubleList xs
```



Your turn: `doubleList : int list -> int list`

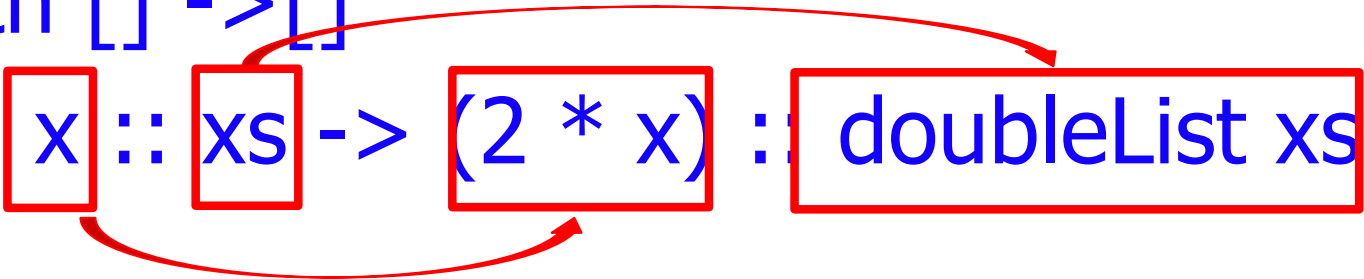
- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

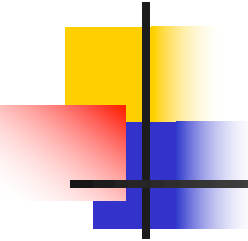
`let rec doubleList list =`

`match list`

`with [] -> []`

`| x :: xs -> (2 * x) :: doubleList xs`





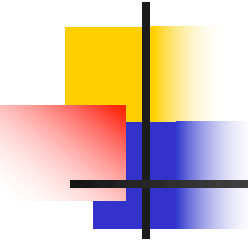
Your turn: `doubleList : int list -> int list`

- Write a function that takes a list of int and returns a list of the same length, where each element has been multiplied by 2

`let rec doubleList list =`

`match list with [] -> []`

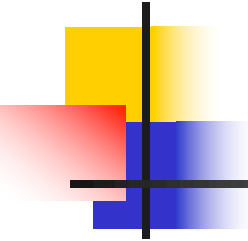
`| (first :: rest) -> (2 * first) :: (doubleList rest)`



Question: Length of list

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

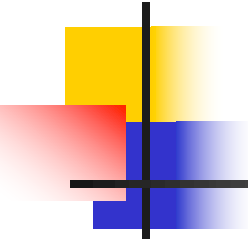
let rec length list =



Question: Length of list

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

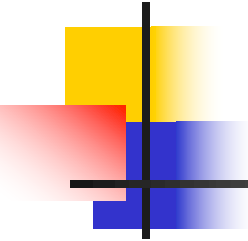
let rec length list =
 match list with



Question: Length of list

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

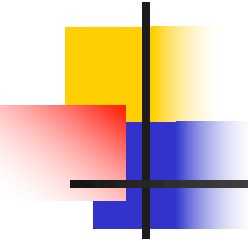
let rec length list =
 match list with



Question: Length of list

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

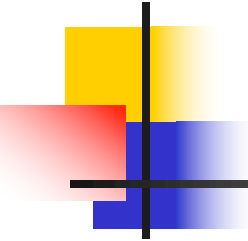
```
let rec length list =  
  match list with [] ->  
    | (a :: bs) ->
```



Question: Length of list

- Problem: write code for the length of the list
 - What result do we give when **list** is empty?

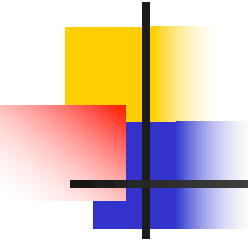
```
let rec length list =  
  match list with [] ->  
    | (a :: bs) ->
```



Question: Length of list

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

```
let rec length list =  
  match list with [] -> 0  
  | (a :: bs) ->
```



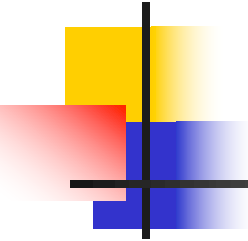
Question: Length of list

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

let rec length list =

match list with [] -> 0

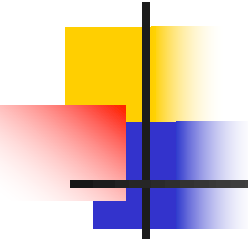
| (a :: bs) -> 1 + length bs



Structural Recursion : List Example

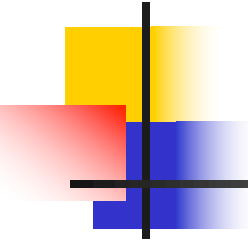
```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
      | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [] is base case
- Cons case recurses on component list **bs**



Same Length

- How can we efficiently answer if two lists have the same length?

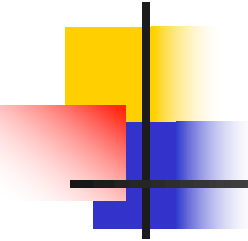


Same Length

- How can we efficiently answer if two lists have the same length?

```
let rec same_length list1 list2 =  
  match list1 with [] ->
```

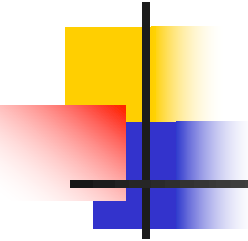
```
  | (x::xs) ->
```



Same Length

- How can we efficiently answer if two lists have the same length?

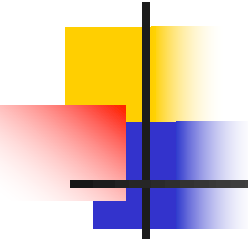
```
let rec same_length list1 list2 =  
  match list1 with [] ->  
    (match list2 with [] -> true  
     | (y::ys) -> false)  
  | (x::xs) ->
```



Same Length

- How can we efficiently answer if two lists have the same length?

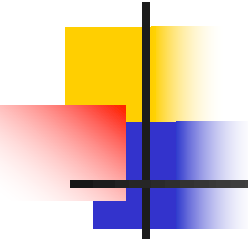
```
let rec same_length list1 list2 =  
  match list1 with [] ->  
    (match list2 with [] ->  
      | (y::ys) ->      )  
  | (x::xs) ->
```



Same Length

- How can we efficiently answer if two lists have the same length?

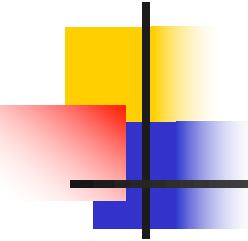
```
let rec same_length list1 list2 =  
  match list1 with [] ->  
    (match list2 with [] -> true  
     | (y::ys) -> false)  
  | (x::xs) ->  
    (match list2 with [] ->  
     | (y::ys) ->
```



Same Length

- How can we efficiently answer if two lists have the same length?

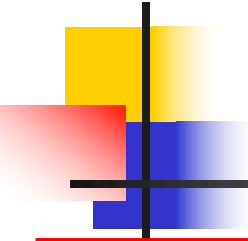
```
let rec same_length list1 list2 =  
  match list1 with [] ->  
    (match list2 with [] -> true  
     | (y::ys) -> false)  
  | (x::xs) ->  
    (match list2 with [] -> false  
     | (y::ys) ->
```



Same Length

- How can we efficiently answer if two lists have the same length?

```
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)
```

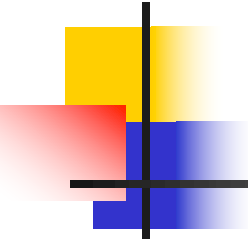


Folding Recursion

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

```
# 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
```

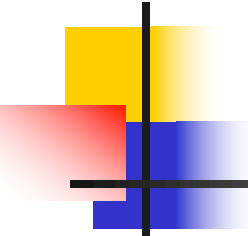
- Computes $(2 * (4 * (6 * 1)))$



Folding Recursion : Length Example

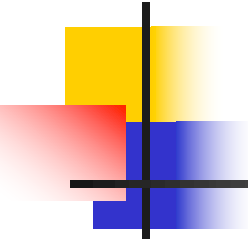
```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
      | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

- Nil case [] is base case, 0 is the base value
- Cons case recurses on component list bs
- What do `multList` and `length` have in common?



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

```
# 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) -> let r = poor_rev xs in r @ [x];;  
val poor_rev : 'a list -> 'a list = <fun>
```

Forward Recursion: Examples

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

Base Case

Operator

Recursive Call

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

Base Case

Operator

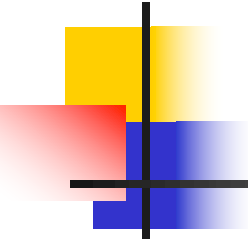
Recursive Call

Recurring over lists



The Primitive
Recursion Fairy

```
# 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 = ()
```



Folding Recursion : Length Example

```
# let rec length list = match list
  with [ ] -> 0   (* Nil case *)
    | a :: bs -> 1 + length bs;; (* Cons case *)
val length : 'a list -> int = <fun>
# let length list =
  fold_right (fun a -> fun r -> 1 + r) list 0;;
val length : 'a list -> int = <fun>
# length [5; 4; 3; 2];;
- : int = 4
```

Forward Recursion: Examples

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

Base Case

Operator

Recursive Call

```
# let double_up =  
  fold_right (fun x -> fun r -> x :: x :: r) list [ ]
```

Operator

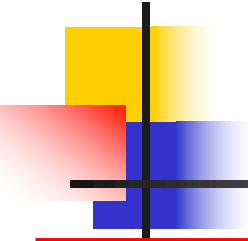
Recursive result

Base Case

```
# double_up ["a";"b"];;  
- : string list = ["a"; "a"; "b"; "b"]
```



- `let rec multList_fr list =
 match list
 with [] -> 1
 | (x::xs) -> let r = (multList_fr xs) in
 (x * r)`



Folding Recursion

- multList folds to the right
- Same as:

```
# let multList list =  
  List.fold_right  
    (fun x -> fun p -> x * p)  
    list 1;;  
  
val multList : int list -> int = <fun>  
  
# multList [2;4;6];;  
- : int = 48
```

Terminology

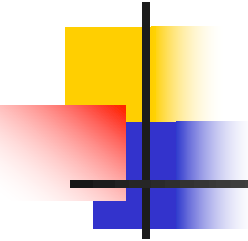
- **Available**: A function call that can be executed by the current expression
- The fastest way to be unavailable is to be guarded by an abstraction (anonymous function, lambda lifted).

- if (h x) then f x else (x + g x)

- if (h x) then (fun x -> f x) else (g (x + x))



Not available



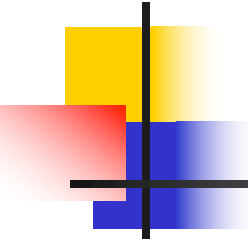
Terminology

- Tail Position: A subexpression s of expressions e , which is **available** and such that if evaluated, will be taken as the value of e (last thing done in this expression)
 - if $(x > 3)$ then $x + 2$ else $x - 4$
 - let $x = 5$ in $x + 4$
- Tail Call: A function call that occurs in tail position
 - if $(h\ x)$ then $f\ x$ else $(x \pm g\ x)$



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 - length

- How can we write length with tail recursion?

let length list =

let rec length_aux list acc_length =

match list

with [] -> acc_length

| (x::xs) ->

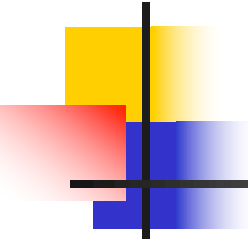
length_aux xs (1 + acc_length)

in length_aux list 0



Your turn: num_neg – tail recursive

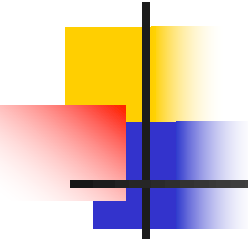
```
# let num_neg list =
```



Your turn: num_neg – tail recursive

```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =
```

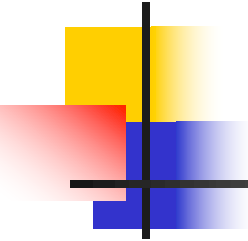
```
    in num_neg_aux ? ?
```



Your turn: num_neg – tail recursive

```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =  
    match list with [] ->  
      | (x :: xs) ->
```

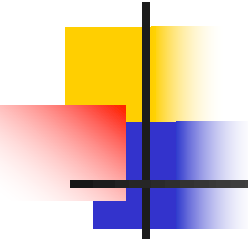
```
in num_neg_aux ? ?
```



Your turn: num_neg – tail recursive

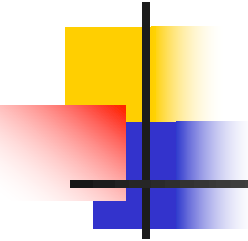
```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =  
    match list with [] -> curr_neg  
    | (x :: xs) ->
```

```
in num_neg_aux ? ?
```



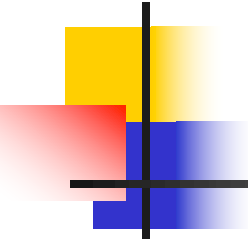
Your turn: num_neg – tail recursive

```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =  
    match list with [] -> curr_neg  
    | (x :: xs) ->  
      num_neg_aux xs ?  
  
  in num_neg_aux ? ?
```



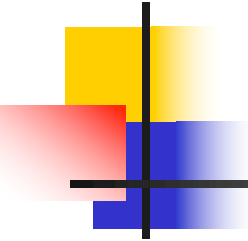
Your turn: num_neg – tail recursive

```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =  
    match list with [] -> curr_neg  
    | (x :: xs) ->  
      num_neg_aux xs  
      (if x < 0 then 1 + curr_neg  
       else curr_neg)  
  in num_neg_aux ? ?
```



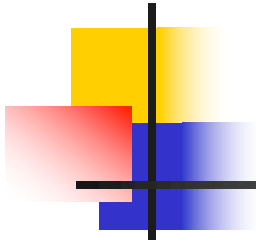
Your turn: num_neg – tail recursive

```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =  
    match list with [] -> curr_neg  
    | (x :: xs) ->  
      num_neg_aux xs  
      (if x < 0 then 1 + curr_neg  
       else curr_neg)  
  in num_neg_aux list ?
```

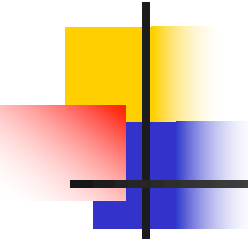


Your turn: num_neg – tail recursive

```
# let num_neg list =  
  let rec num_neg_aux list curr_neg =  
    match list with [] -> curr_neg  
    | (x :: xs) ->  
      num_neg_aux xs  
      (if x < 0 then 1 + curr_neg  
       else curr_neg)  
  in num_neg_aux list 0
```



```
let num_neg list =  
List.fold_left  
  (fun curr_neg -> (fun x ->  
    (if x < 0 then 1 + curr_neg else curr_neg)  
  )  
  )  
  0  
  list
```



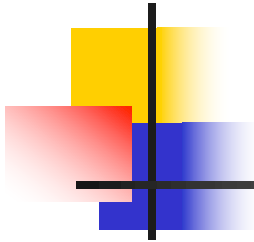
Folding

```
# 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>
```

$\text{fold_left } f \ a \ [x_1; x_2; \dots; x_n] = f(\dots(f(f \ a \ x_1) \ x_2) \dots) x_n$

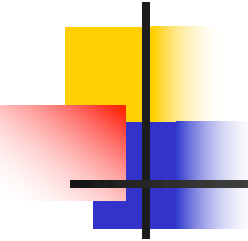
```
# let rec fold_right f list b = match list
  with [] -> b | (x :: xs) -> f x (fold_right f xs b);;
val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b =
  <fun>
```

$\text{fold_right } f \ [x_1; x_2; \dots; x_n] \ b = f \ x_1(f \ x_2(\dots(f \ x_n \ b) \dots))$



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



Mapping Recursion

```
# 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]
```

Map is forward recursive

```
# let rec map f list =
```

```
  match list
```

```
  with [] -> []
```

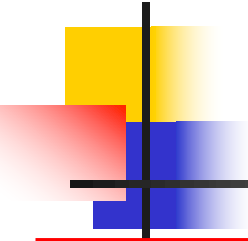
```
  | (h::t) -> (f h) :: (map f t);;
```

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

```
# let map f list =
```

```
  List.fold_right (fun h -> fun r -> (f h) :: r)
                  list [];;
```

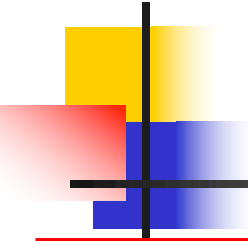
```
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```



Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# 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]
```



Mapping Recursion

- Can use the higher-order recursive map function instead of direct recursion

```
# 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 explicit recursion