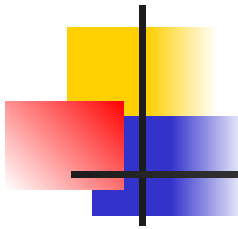


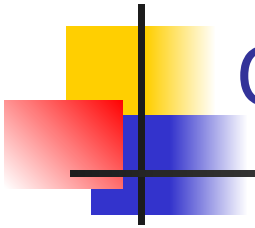
Programming Languages and Compilers (CS 421)

Talia Ringer (they/them)

4218 SC, UIUC

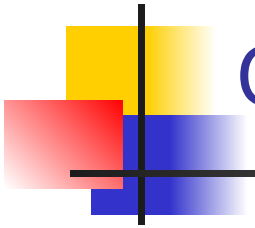
https://courses.grainger.illinois.edu/cs421/fa2023/

Based heavily on slides by Elsa Gunter, which were based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha



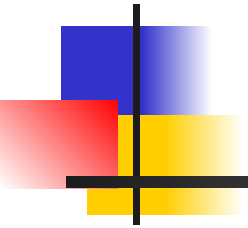
Objectives for Today

- On Thursday, we took an in depth look at **lists** and **recursive functions** defined over lists.
- We also previewed some common **higher-order functions** over lists—map and fold.
- Today, we will look at these higher-order functions in more detail, looking at the difference between **folding left** and **folding right**.
- We will also learn about **forward recursion** and **tail recursion**, and how these relate to folding left and folding right.

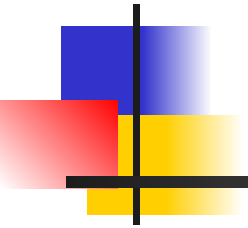


Objectives for Today

- On Thursday, we took an in depth look at **lists** and **recursive functions** defined over lists.
- We also previewed some common **higher-order functions** over lists—map and fold.
- Today, we will look at these higher-order functions in more detail, looking at the difference between **folding left** and **folding right**.
- We will also learn about **forward recursion** and **tail recursion**, and how these relate to folding left and folding right.



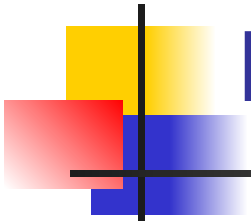
Questions from last time?



Forward Recursion

Forward Recursion

- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer
- Corresponds to **folding right** (with caveats)



Forward Recursion

- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer
- Corresponds to **folding right** (with caveats)

Forward Recursion

- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer
- Corresponds to **folding right** (with caveats)

Forward Recursion

- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer
- Corresponds to **folding right** (with caveats)

There are two different orders we can fold over lists in—we'll see the other one later in class.

Forward Recursion by fold_right

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


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

Forward Recursion by fold_right

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

base case / id **operator** **recursion (first)**

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

recursion (first) **operator** **base case / id**

Forward Recursion

Forward Recursion by fold_right

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

Forward Recursion by **fold_right**

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

base case / id

operator

recursion (first)

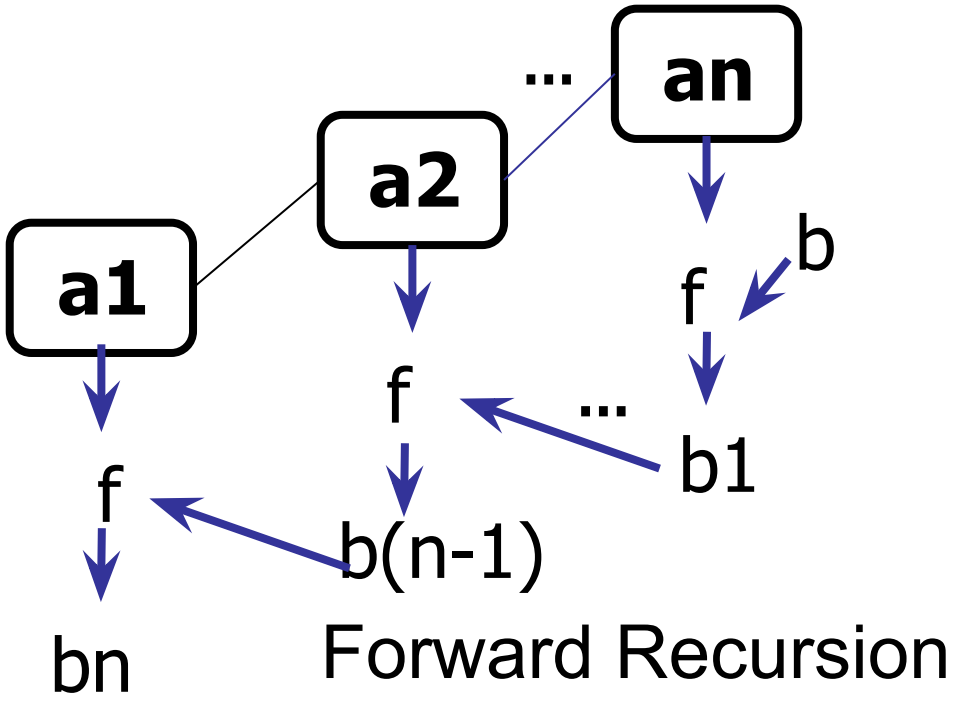
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```

Forward Recursion

Forward Recursion by **fold_right**

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

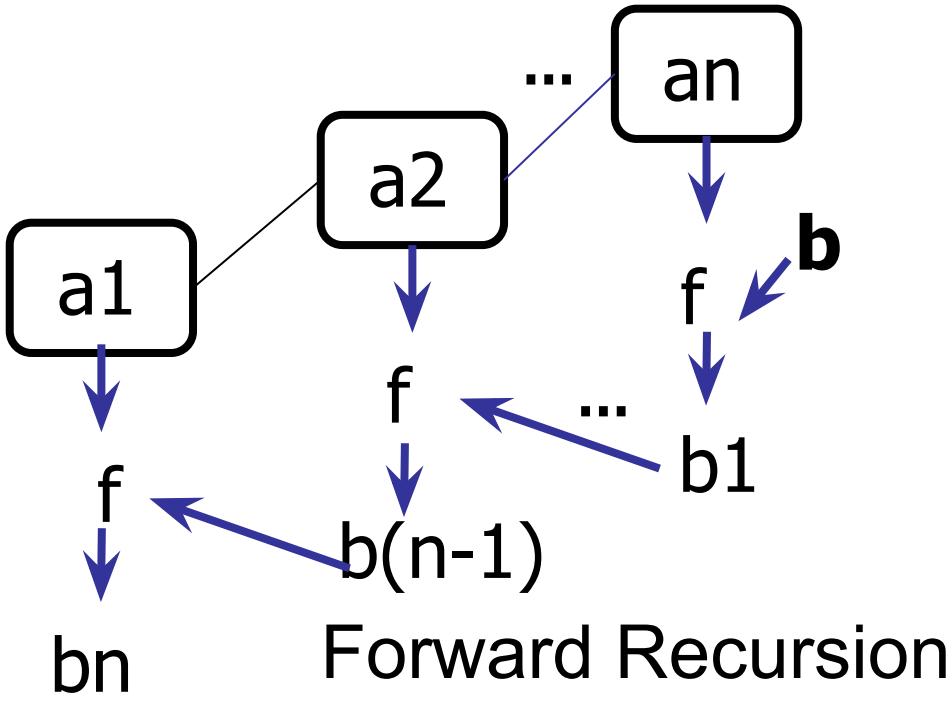
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```



Forward Recursion by **fold_right**

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

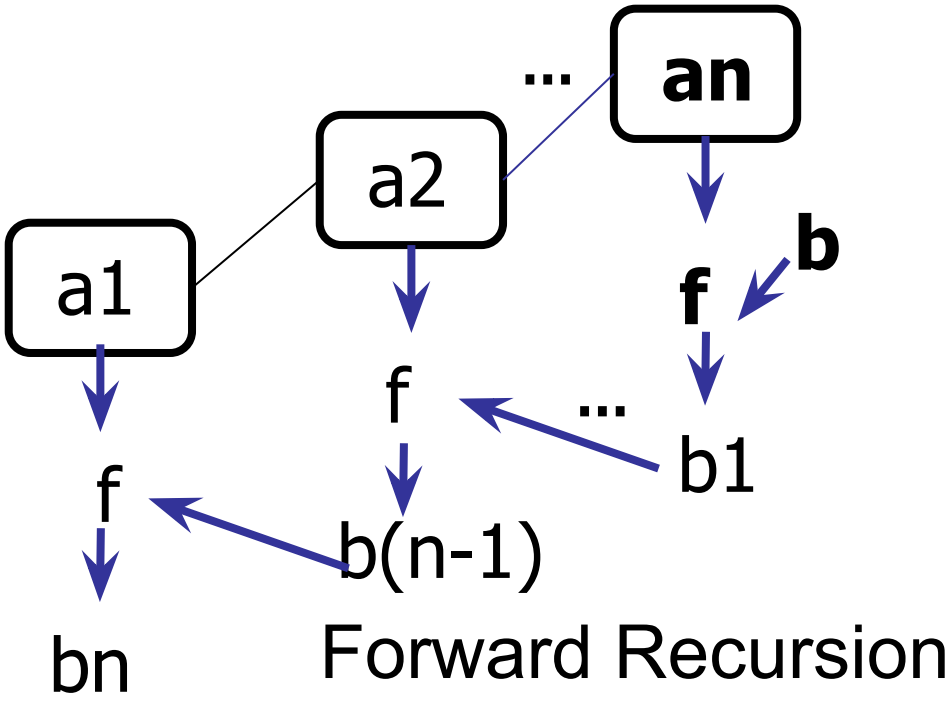
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```



Forward Recursion by **fold_right**

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

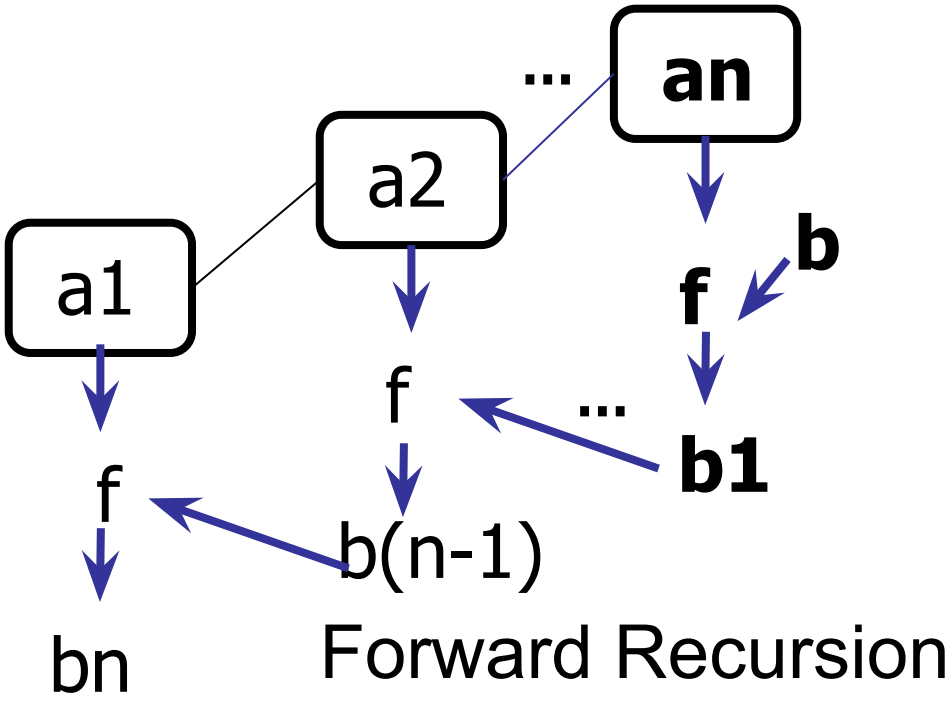
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```



Forward Recursion by **fold_right**

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

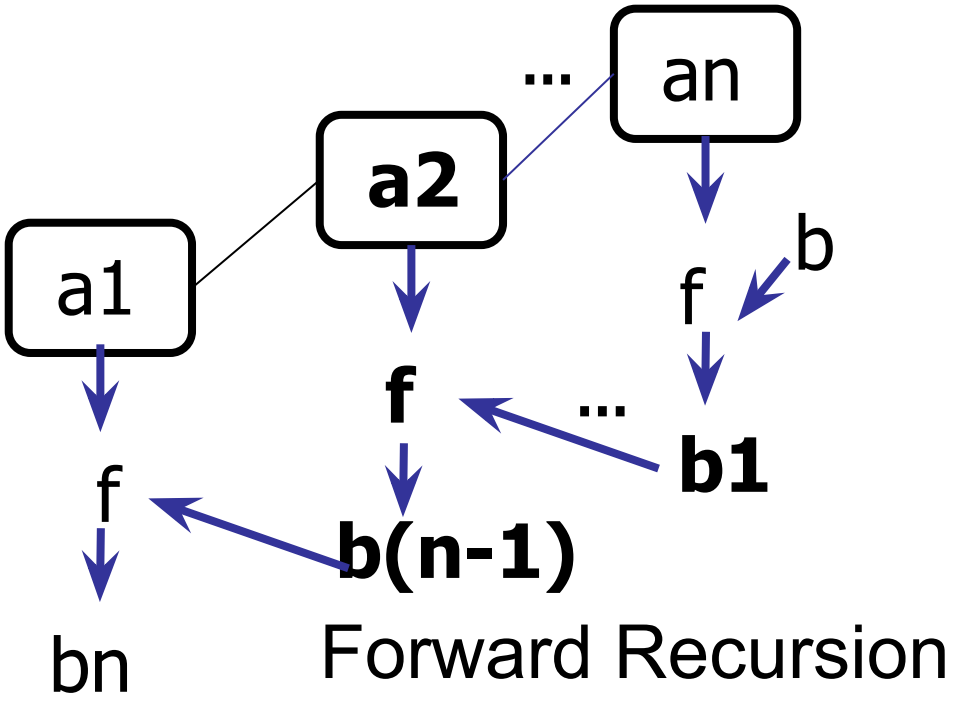
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```



Forward Recursion by **fold_right**

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

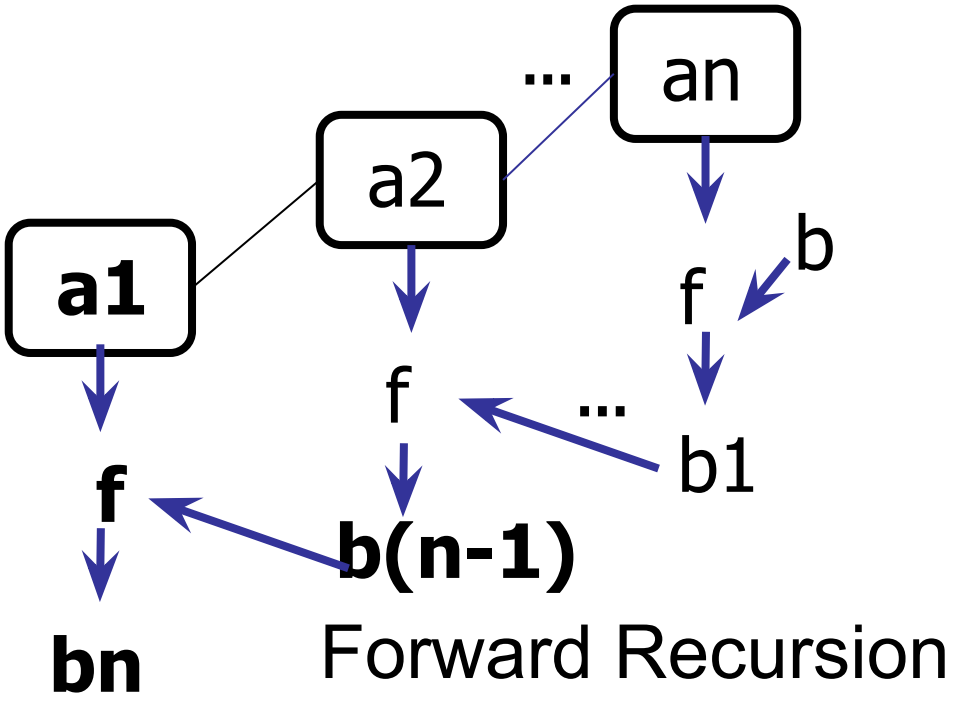
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```



Forward Recursion by **fold_right**

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

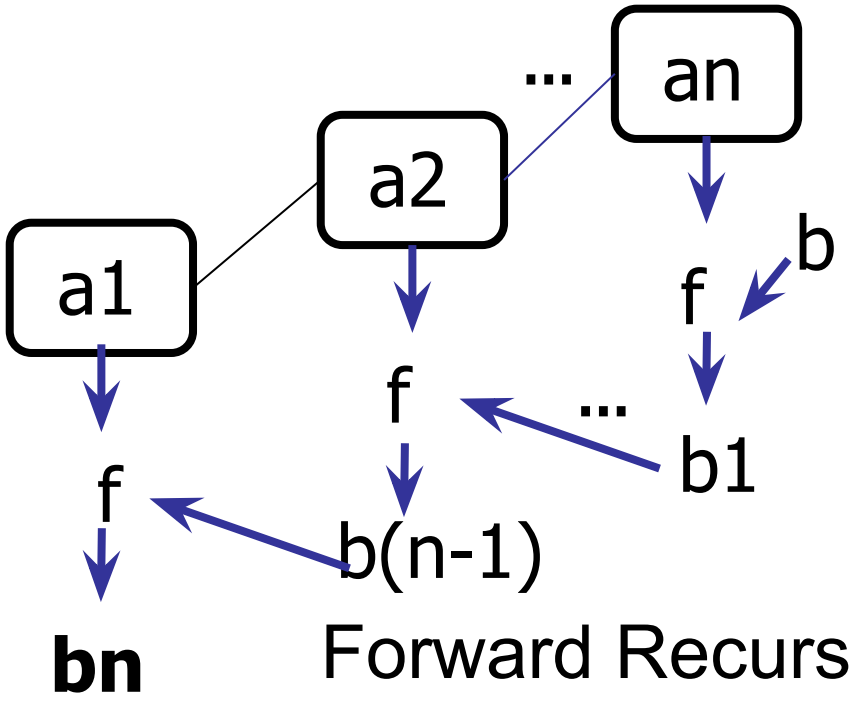
```
val fold_right :
  ('a -> 'b -> 'b) ->
  'a list ->
  'b ->
  'b
= <fun>
```



Forward Recursion by **fold_right**

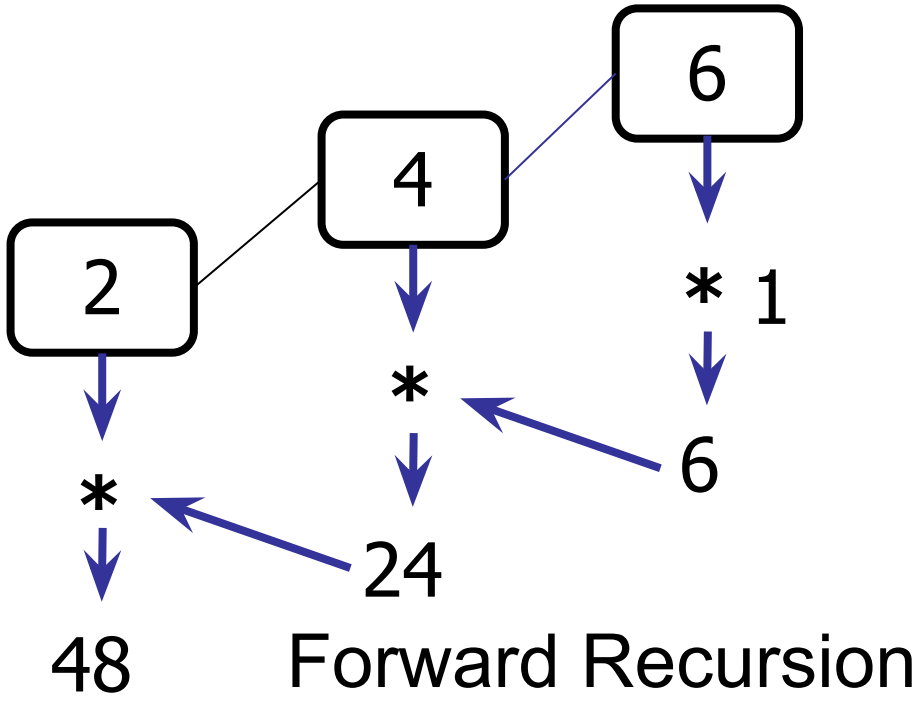
```
# let rec fold_right f list b =
    match list with
    | [ ] -> b
    | (x :: xs) -> f x (fold_right f xs b);;
```

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



Forward Recursion by fold_right

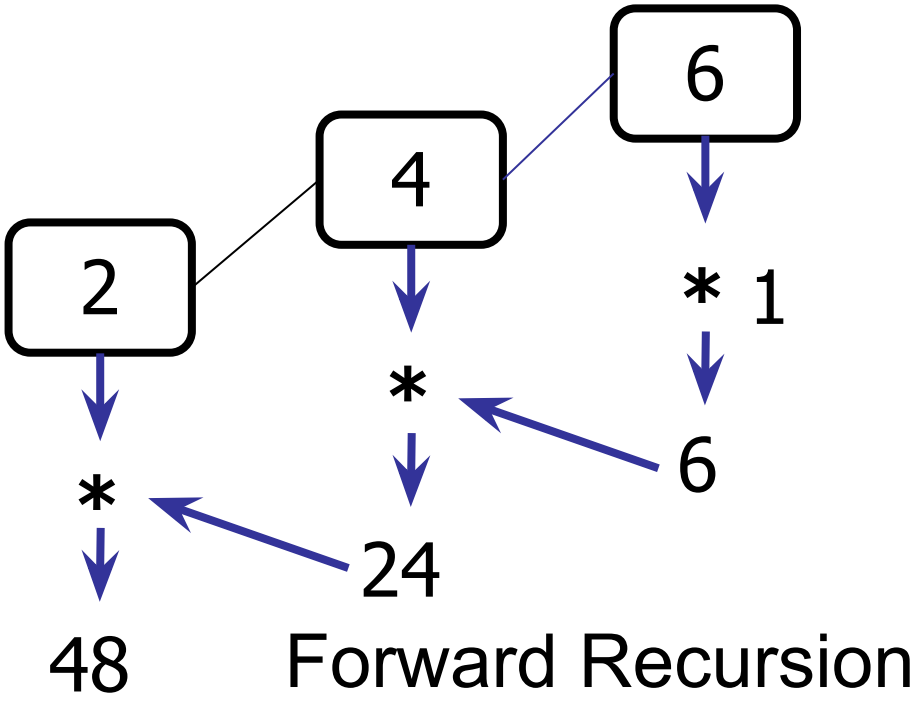
```
let rec multList list =
  match list with
  | [ ] -> 1
  | x :: xs -> x * multList xs;;
```



Forward Recursion by fold_right

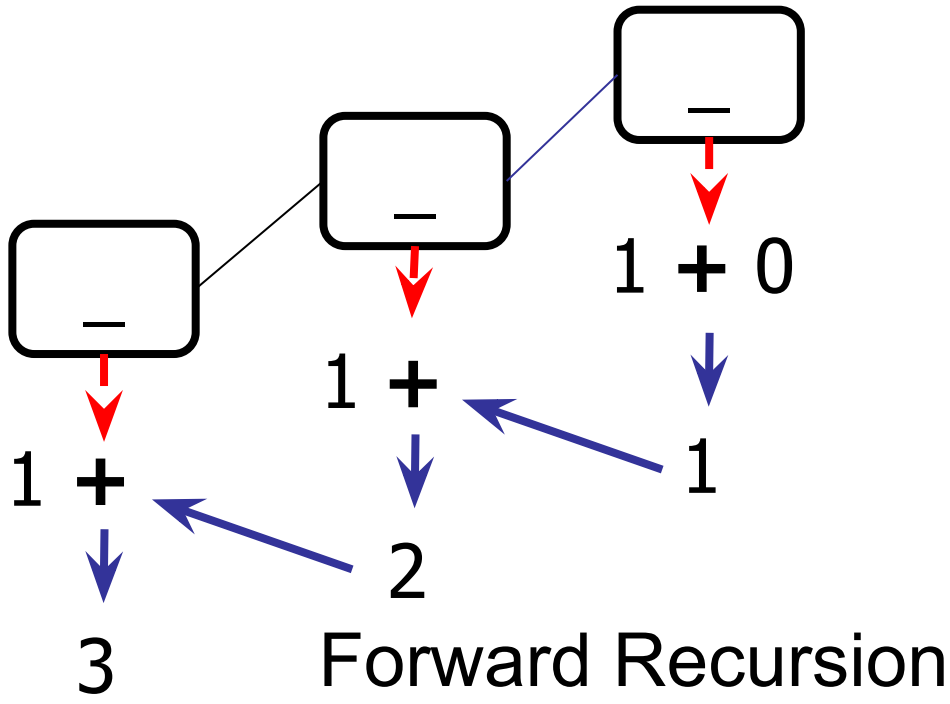
```
let rec multList list =
  match list with
  | [ ] -> 1
  | x :: xs -> x * multList xs;;
```

```
let multList list =
  List.fold_right
    (fun x p -> x * p)
    list
    1;;
```



Forward Recursion by fold_right

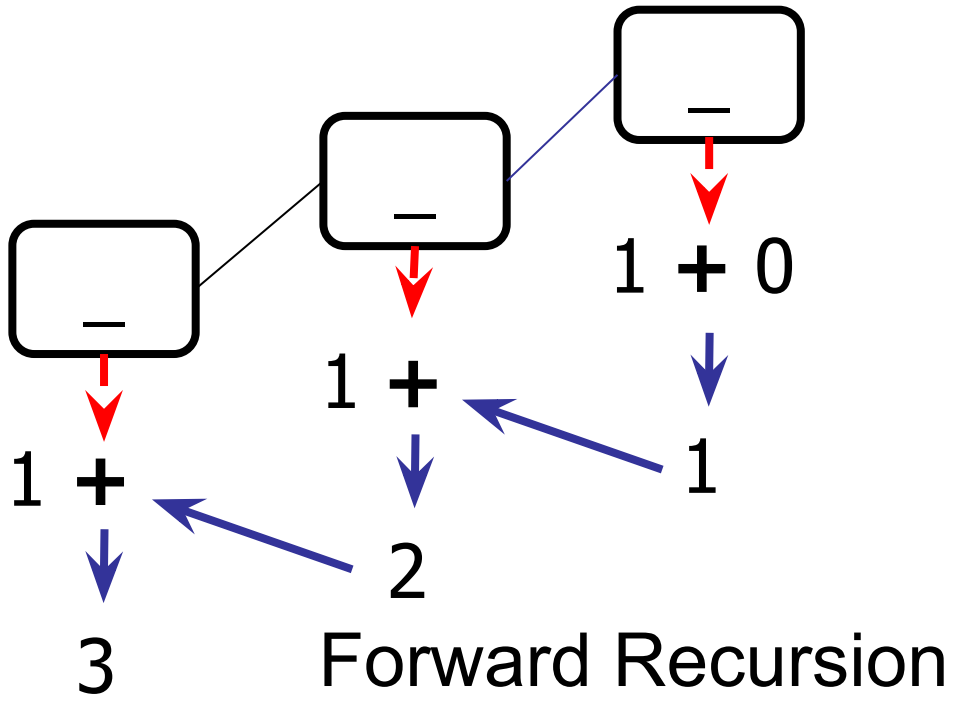
```
let rec length list =
  match list with
  | [ ] -> 0
  | _ :: bs -> 1 + length bs;;
```



Forward Recursion by fold_right

```
let rec length list =
  match list with
  | [ ] -> 0
  | _ :: bs -> 1 + length bs;;
```

```
let length list =
  List.fold_right
    (fun _ r -> 1 + r)
    list
    0;;
```



Forward Recursion by fold_right

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

Forward Recursion by fold_right

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

base case / id — **operator** — **recursion (first)**

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

recursion (first) — **operator** — **base case / id**

Forward Recursion

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> ?) list1 ?;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

append [4; 5; 6] [1; 2; 3] = [4; 5; 6; 1; 2; 3]
append [ ] [1; 2; 3] = [1; 2; 3]
append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> ?) list1 ?;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

append [4; 5; 6] [1; 2; 3] = [4; 5; 6; 1; 2; 3]
append **[ ]** [1; 2; 3] = [1; 2; 3]
append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> ?) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

append [4; 5; 6] [1; 2; 3] = [4; 5; 6; 1; 2; 3]
append [ ] **[1; 2; 3]** = **[1; 2; 3]**
append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> ?) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

append **[4; 5; 6]** [1; 2; 3] = [4; 5; 6; 1; 2; 3]
append [ ] [1; 2; 3] = [1; 2; 3]
append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> a :: ?) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

append [**4**; 5; 6] [1; 2; 3] = [**4**; 5; 6; 1; 2; 3]
append [ ] [1; 2; 3] = [1; 2; 3]
append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> a :: ?) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

append [**4**; **5; 6**] [**1; 2; 3**] = [**4; 5; 6; 1; 2; 3**]
append [ ] [1; 2; 3] = [1; 2; 3]
append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

```
# let append list1 list2 =
    List.fold_right (fun a r -> a :: ?) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
```

4 :: append [5; 6] [1; 2; 3] = [**4; 5; 6; 1; 2; 3**]

append [ ] [1; 2; 3] = [1; 2; 3]

append [1; 2] [ ] = [1; 2]

Forward Recursion by **fold_right**

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

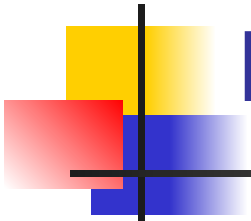
4 :: append [5; 6] [1; 2; 3] = [**4; 5; 6; 1; 2; 3**]

append [ ] [1; 2; 3] = [1; 2; 3]

append [1; 2] [ ] = [1; 2]

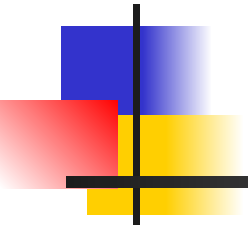
Forward Recursion by **fold_right**

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

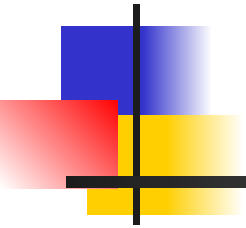


Forward Recursion

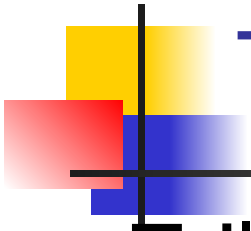
- Forward Recursion form of **Structural Recursion** (recurse on substructures)
- In **forward** recursion, **first call the function recursively** on all recursive components, and then **build final result**
- **Wait** until whole structure has been traversed **to start building** answer
- Corresponds to **folding right** (with caveats)



Questions so far?

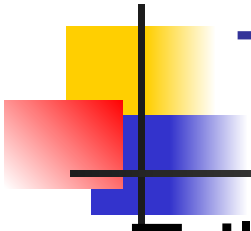


Tail Recursion



Tail Recursion

- Tail Recursion form of **Structural Recursion** (recurse on substructures)
- In **tail** recursion, **first build the intermediate result**, then **call the function recursively**
- Build answer **as you go**, typically using an **accumulator** or **auxiliary function**
- Corresponds to **folding left** (with caveats)



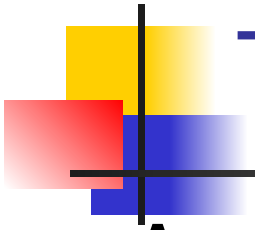
Tail Recursion

- Tail Recursion form of **Structural Recursion** (recurse on substructures)
- In **tail** recursion, **first build the intermediate result**, then **call the function recursively**
- Build answer **as you go**, typically using an **accumulator** or **auxiliary function**
- Corresponds to **folding left** (with caveats)

Tail Recursion

- Tail Recursion form of **Structural Recursion** (recurse on substructures)
- In **tail** recursion, **first build the intermediate result**, then **call the function recursively**
- Build answer **as you go**, typically using an **accumulator** or **auxiliary function**
- Corresponds to **folding left** (with caveats)

Soon we'll see the other direction we can fold in.

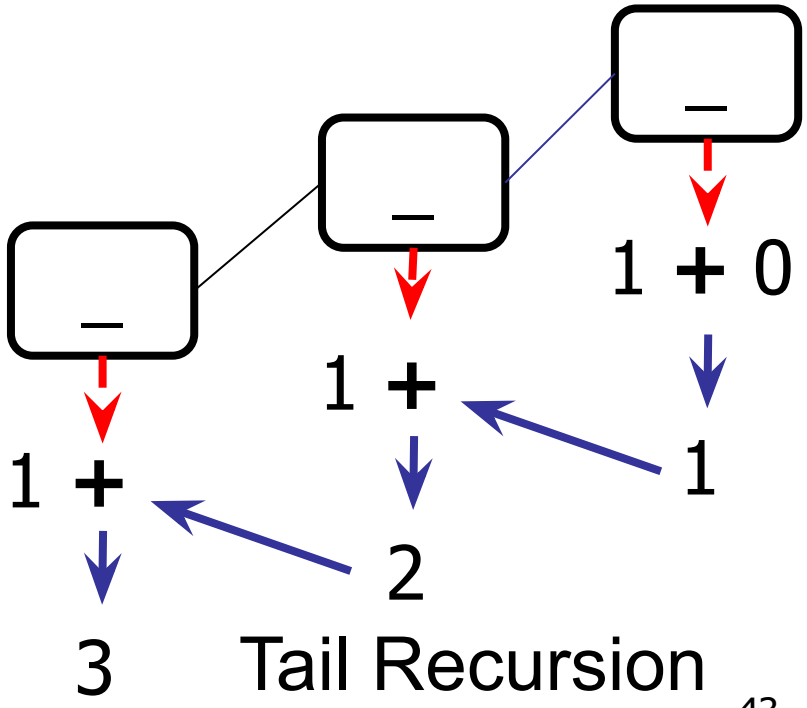


Tail Recursion

- 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

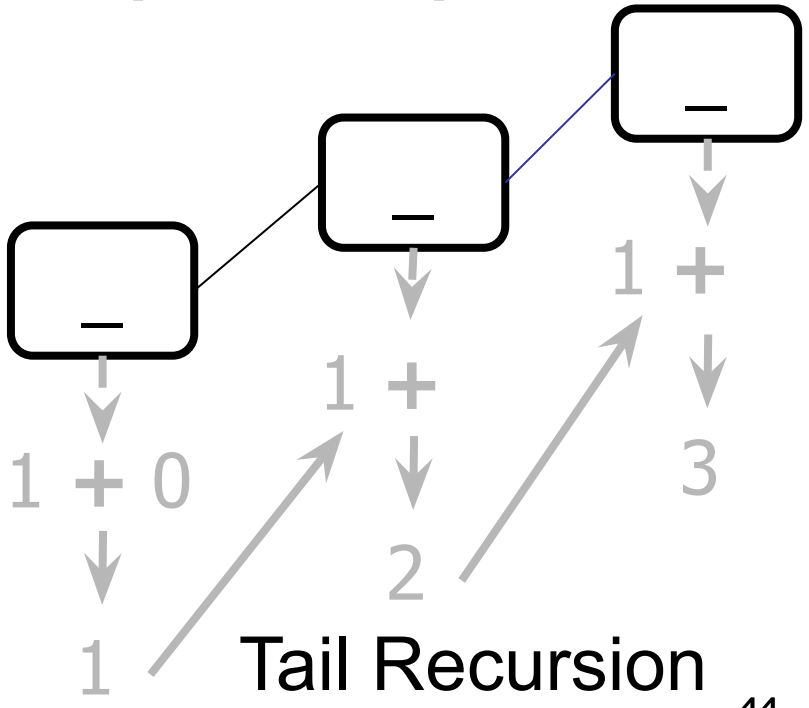
Forward Recursion - Length

```
let rec length list =
  match list with
  | [ ] -> 0
  | _ :: bs -> 1 + length bs;;
```



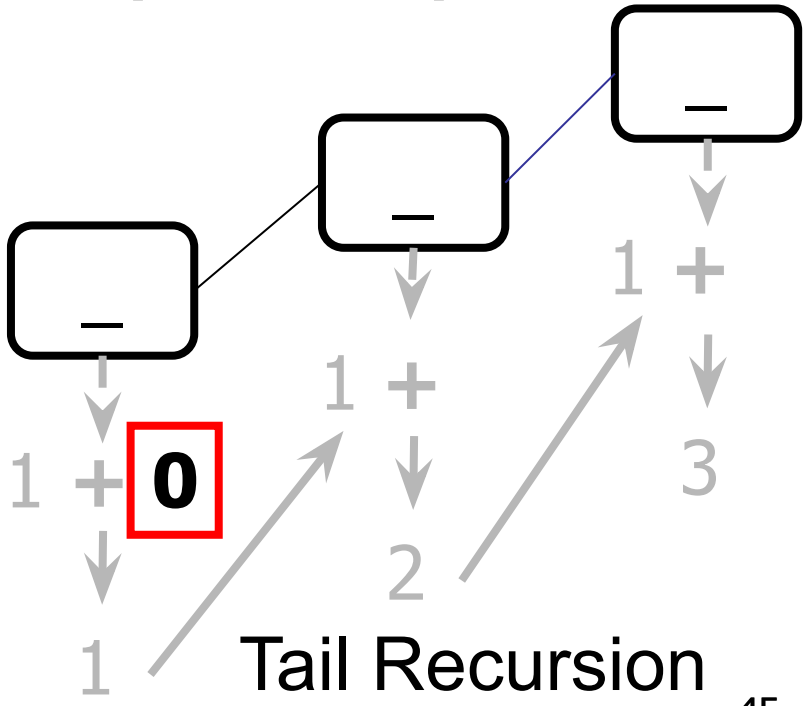
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
   | [ ] -> acc
   | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



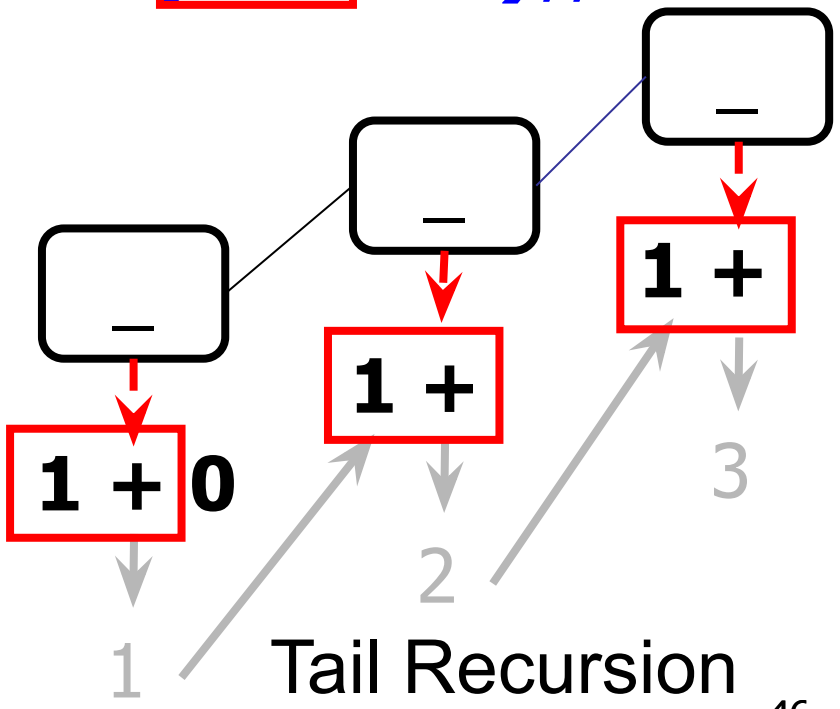
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
   | [ ] -> acc
   | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



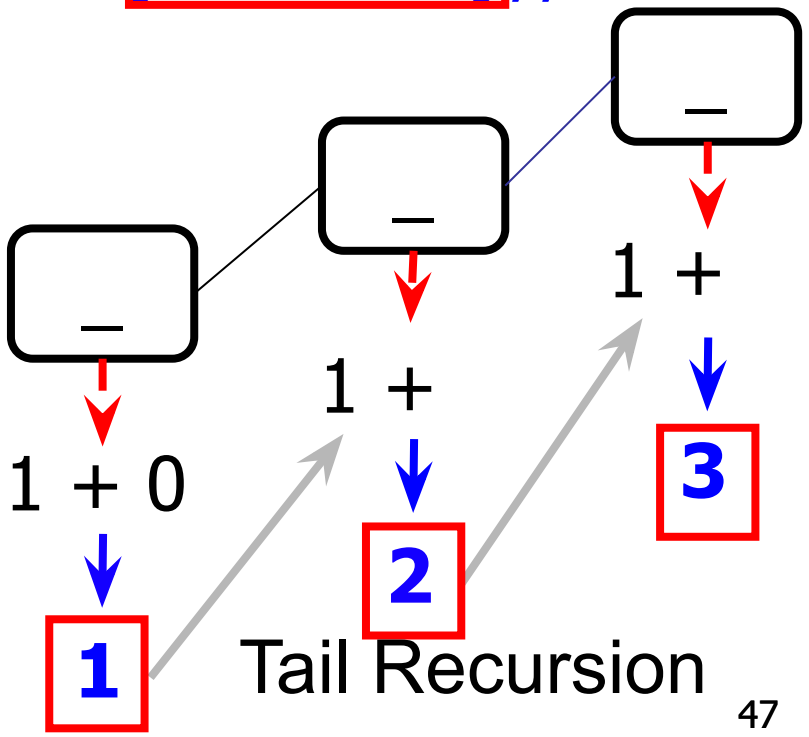
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



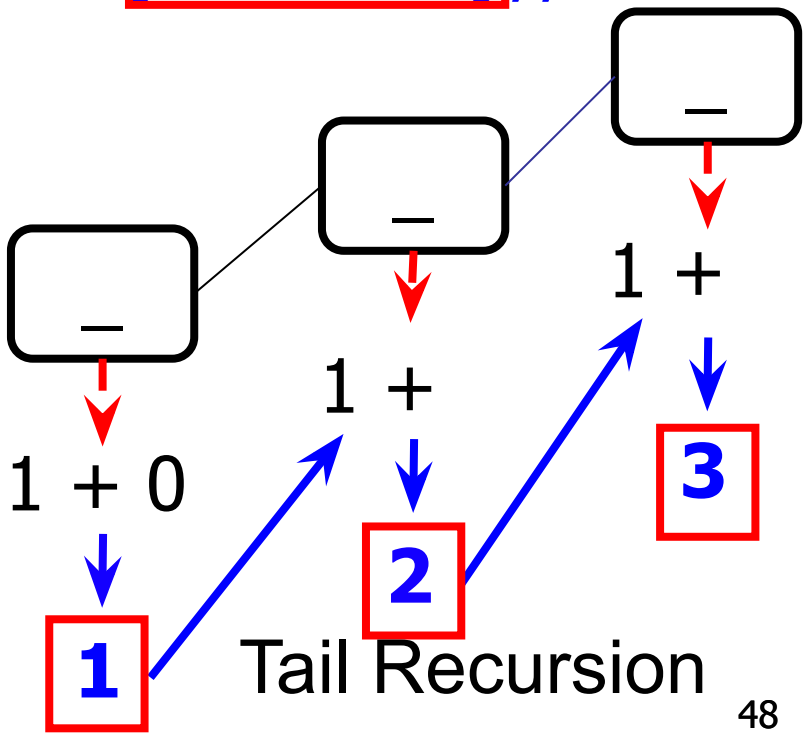
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



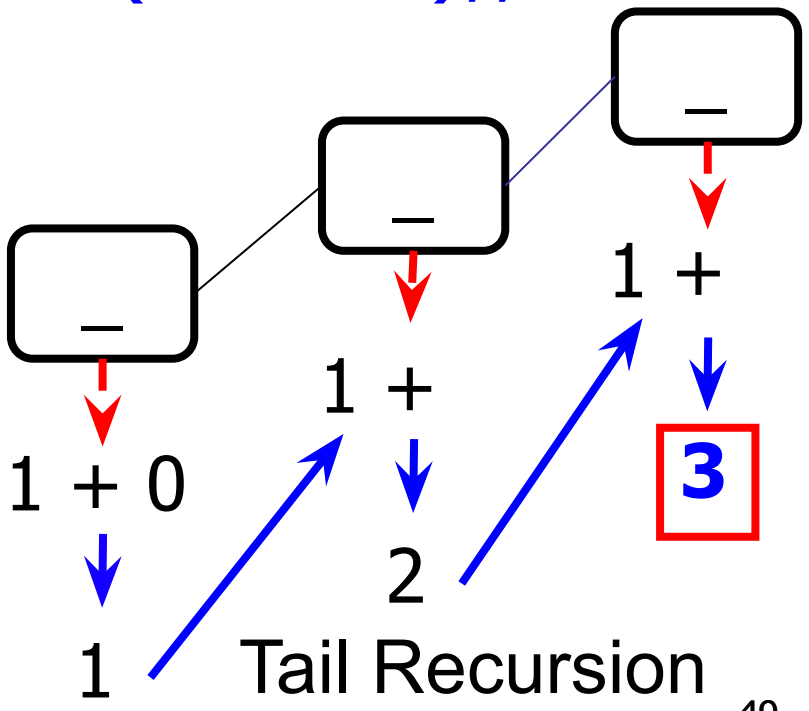
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



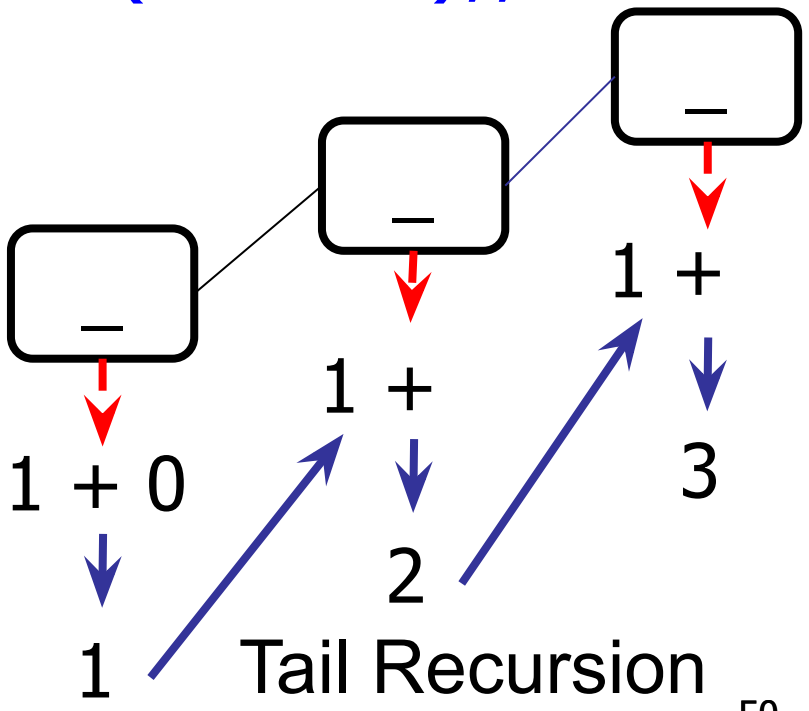
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



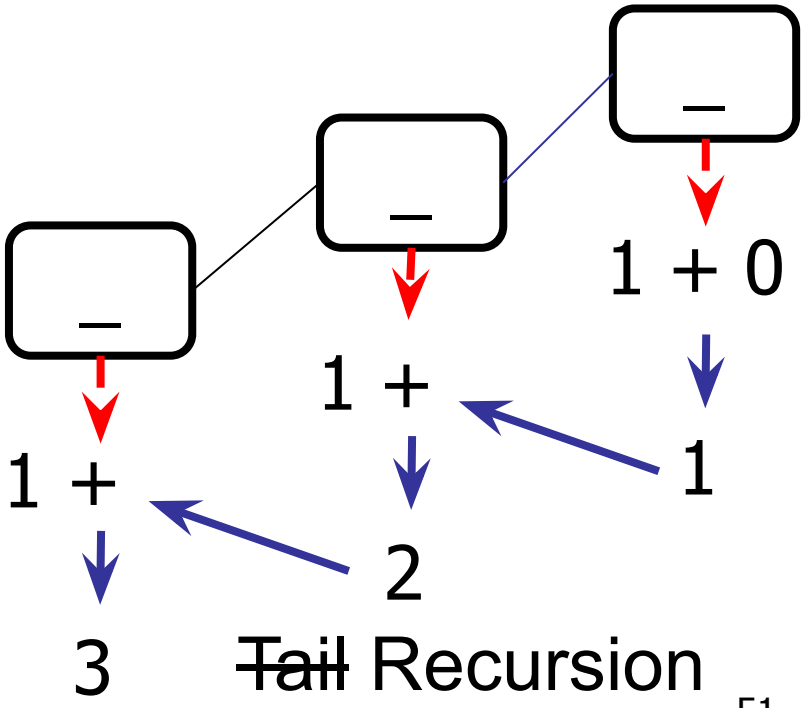
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```



Forward Recursion - Length

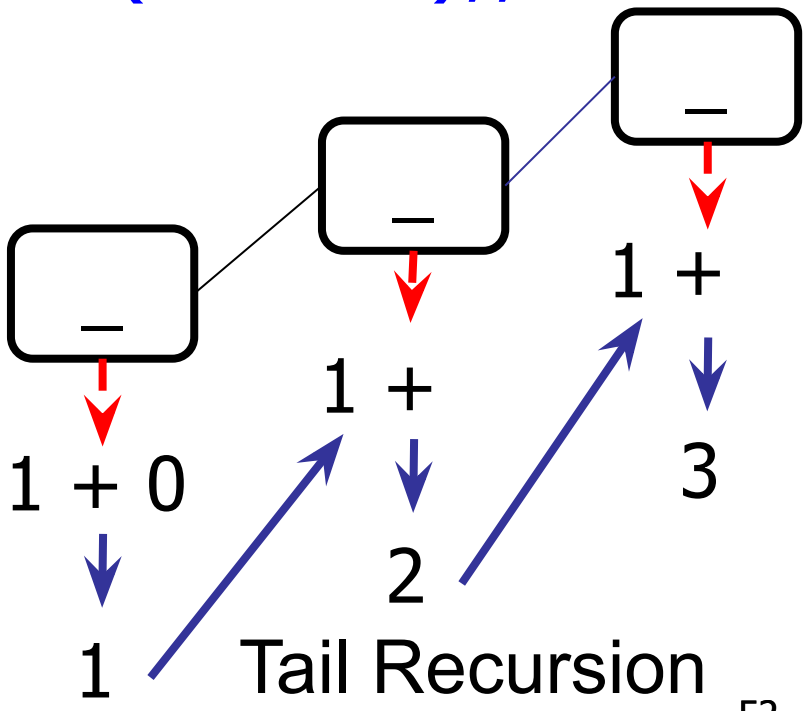
```
let rec length list =
  match list with
  | [ ] -> 0
  | _ :: bs -> 1 + length bs;;
```

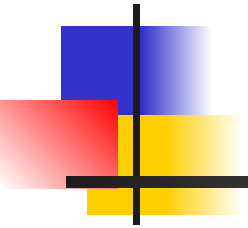


~~Tail~~ Recursion

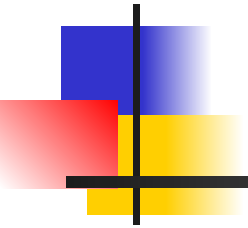
Tail Recursion - Length

```
let rec length_aux list acc =
  match list with
  | [ ] -> acc
  | _ :: bs -> length_aux bs (1 + acc);;

let length =
  length_aux list 0;;
```





Questions so far?



Forward vs. Tail Recursion: Runtime

Forward vs. Tail Recursion

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

# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;
```

What is the runtime of each function?

Runtime

Forward vs. Tail Recursion

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

# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;
```

What is the runtime of each function?

Runtime

Forward vs. Tail Recursion

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

# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;
```

What is the runtime of each function?

Runtime

Forward vs. Tail Recursion

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

# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;
```

What is the runtime of each function?

Runtime

Forward vs. Tail Recursion

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

# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;
```

What is the runtime of each function?

Runtime

Forward vs. Tail Recursion

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

# let rec rev_aux list revlist =
  match list with
  | [ ] -> revlist
  | x :: xs -> rev_aux xs (x::revlist);;
# let rev list = rev_aux list [ ];;
```

What is the runtime of each function?

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. Tail Recursion

- 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]

Runtime

Forward vs. **Tail Recursion**

- rev [1;2;3] =
- rev_aux [1;2;3] [ ] =
- rev_aux [2;3] [1] =
- rev_aux [3] [2;1] =
- rev_aux [ ] [3;2;1] = [3;2;1]

Forward vs. **Tail Recursion**

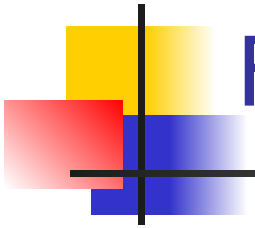
- rev [1;2;3] =
- rev_aux [1;2;3] [ ] =
- rev_aux [2;3] [1] =
- rev_aux [3] [2;1] =
- rev_aux [ ] [3;2;1] = [3;2;1]

Forward vs. **Tail Recursion**

- rev [1;2;3] =
- rev_aux [1;2;3] [ ] =
- rev_aux [2;3] [1] =
- rev_aux [3] [2;1] =
- rev_aux [ ] [3;2;1] = [3;2;1]

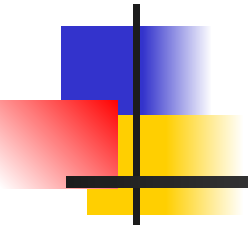
Forward vs. **Tail Recursion**

- rev [1;2;3] =
- rev_aux [1;2;3] [ ] =
- rev_aux [2;3] [1] =
- rev_aux [3] [2;1] =
- rev_aux [ ] [3;2;1] = [3;2;1]



Forward vs. **Tail Recursion**

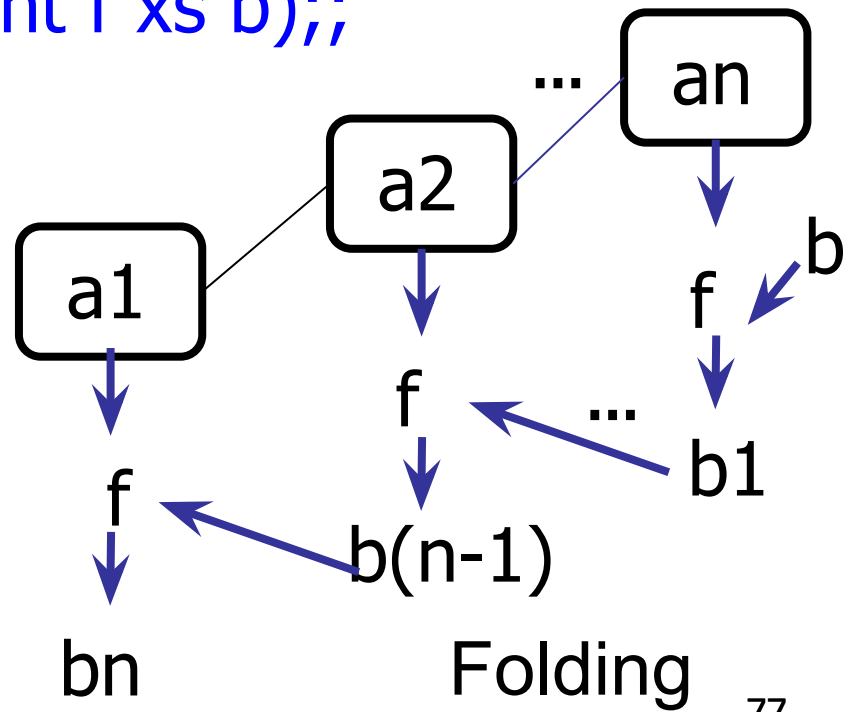
- rev [1;2;3] =
- rev_aux [1;2;3] [ ] =
- rev_aux [2;3] [1] =
- rev_aux [3] [2;1] =
- rev_aux [ ] [3;2;1] = [3;2;1]



Folding: Right vs. Left

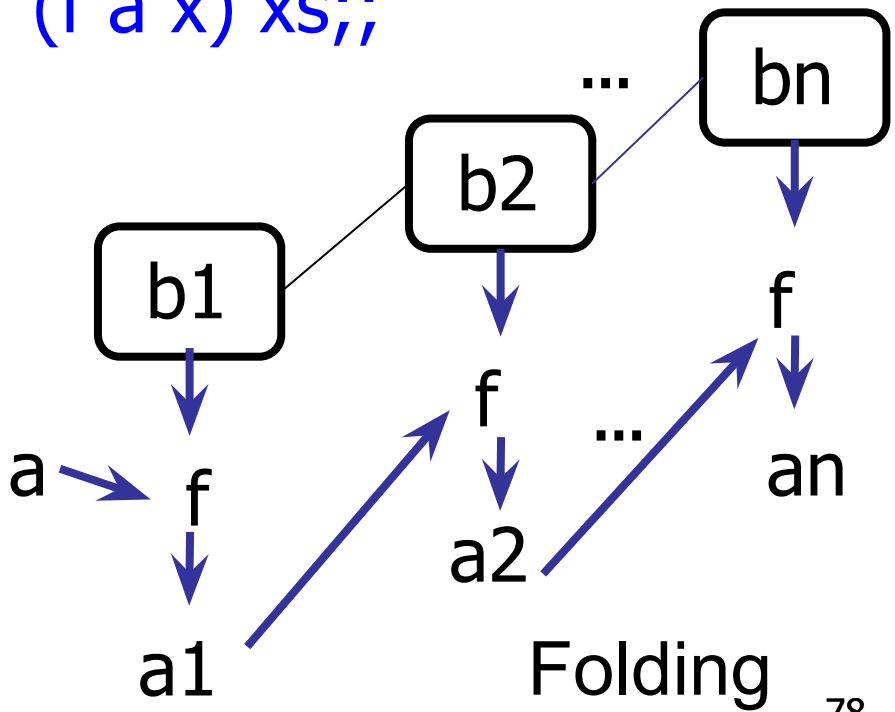
Forward Recursion by fold_right

```
# 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>
```



Tail Recursion by fold_left

```
# 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>
```



Folding Left vs. Folding Right

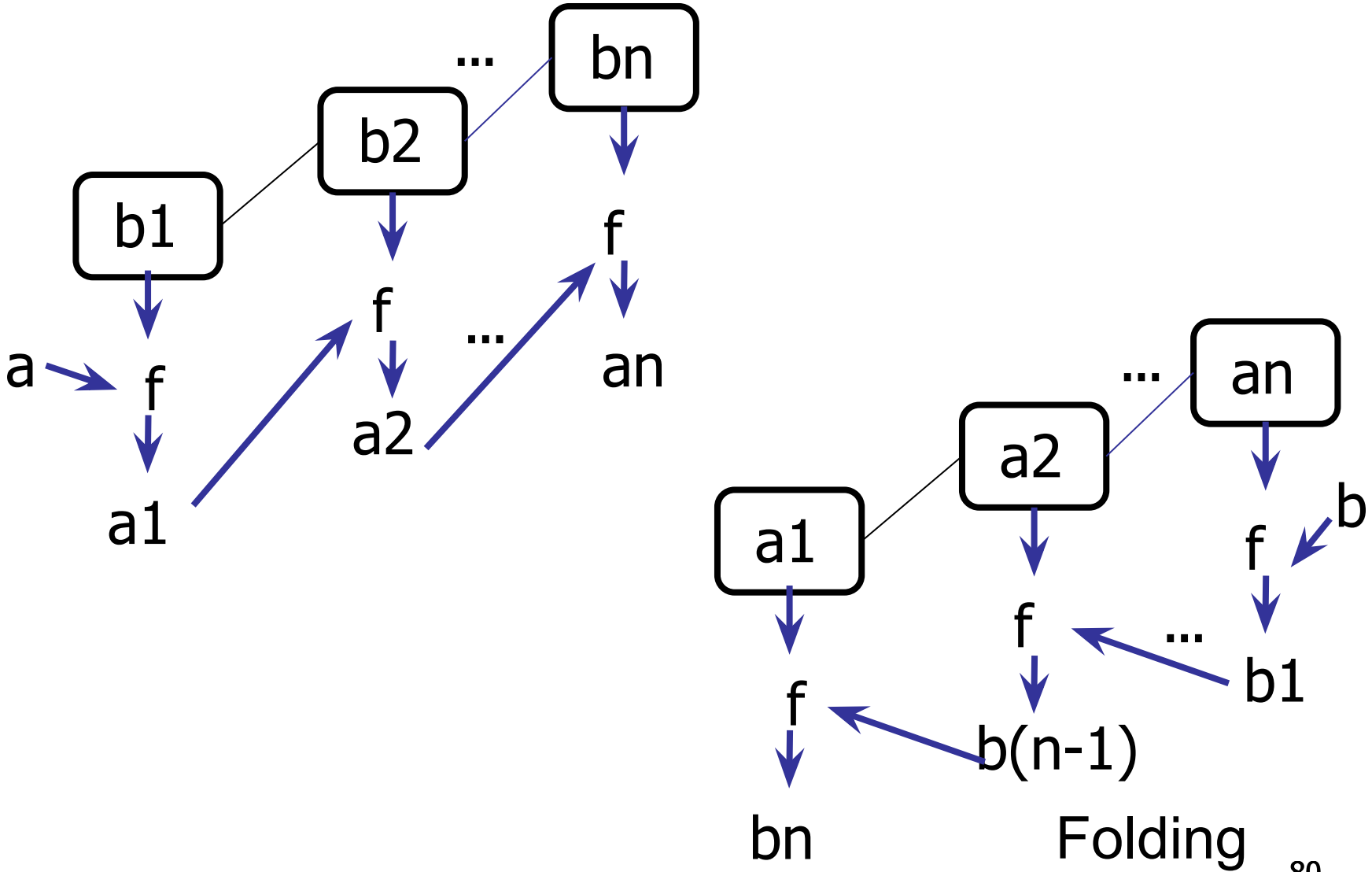
```
# let rec fold_left f a list =
   match list with
   | [] -> a
   | (x :: xs) -> fold_left f (f a x) xs;;
```

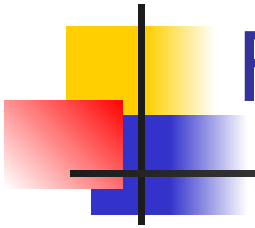
fold_left f a [x_1; x_2;…;x_n] = f (… (f (f a x_1) x_2)…) x_n

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

fold_right f [x_1; x_2;…;x_n] b = f x_1(f x_2 (…(f x_n b)…))

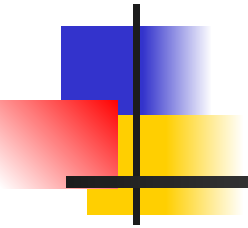
Folding Left vs. Folding Right



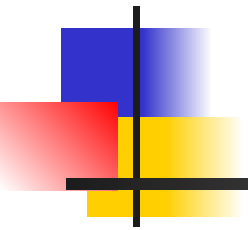


Folding

- Can replace recursion by **fold_right** in any **forward primitive recursive definition**
  - Primitive recursive means it **recurses only on immediate subcomponents** of recursive data structure (like the tail of a list)
- Can replace recursion by fold_left in any **tail primitive recursive definition**



Questions?



Next Class: Continuation-Passing Style

Reminders

- **Quiz 2** on **MP3** next **Tuesday**
- **Midterm 1** in **CBTF** 9/14-9/16—**please sign up!**
- All deadlines can be found on **course website**
- Use **office hours** and **class forums** for help
- **Please thank Elsa again for covering <3**

Next Class