Programming Languages and Compilers (CS 421)

Sasa Misailovic

4110 SC, UIUC

https://courses.engr.illinois.edu/cs421/fa2024/CS421C

Based on slides by Elsa Gunter, which are based in part on previous slides by Mattox Beckman and updated by Vikram Adve and Gul Agha

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

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

Forward Recursion

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

Question

- How do you write length with forward recursion?

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

Question

- How do you write length with forward recursion?

```
let rec length l =
    match l with [] -> 0
     | (a :: bs) -> let t = length bs
                    in 1 + t
```

Mapping Recursion

- One common form of structural recursion applies a function to each element in the structure

```
# 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]
```

Mapping Functions Over Lists

```
# 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]
```

Mapping Recursion

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

- 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>
```

- Same function, but no rec

Your turn now

Write a function

`make_app : ((‘a -> ‘b) * ‘a) list -> ‘b list`

that takes a list of function – input pairs and gives the result of applying each function to its argument. Use map, no explicit recursion.

```
let make_app lst =
```

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 Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
      [ ] -> 0
    | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

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

# prodlist [2;3;4];;
- : int = 24
```

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
      [ ] -> 0
    | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

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

# prodlist [2;3;4];;
- : int = 24
```

Base Case

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
      [ ] -> 0
    | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

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

# prodlist [2;3;4];;
- : int = 24
```

Recursive Call

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
      [ ] -> 0
    | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

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

# prodlist [2;3;4];;
- : int = 24
```

Head Element

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumlist list = match list with
      [ ] -> 0
    | x::xs -> x + sumlist xs;;

# sumlist [2;3;4];;
- : int = 9

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

# prodlist [2;3;4];;
- : int = 24
```

Combining Operator

Recursing over lists

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

# fold_right
    (fun val init -> val + init)
    [1; 2; 3]
    0;;
- : int = 6
```

```
                    3rd      2nd             1st
Order:               1  + (   2   +  ( 3 + 0 ) )
```

`Watch for parentheses: deeper nested is evaluated first`

Recursing over lists

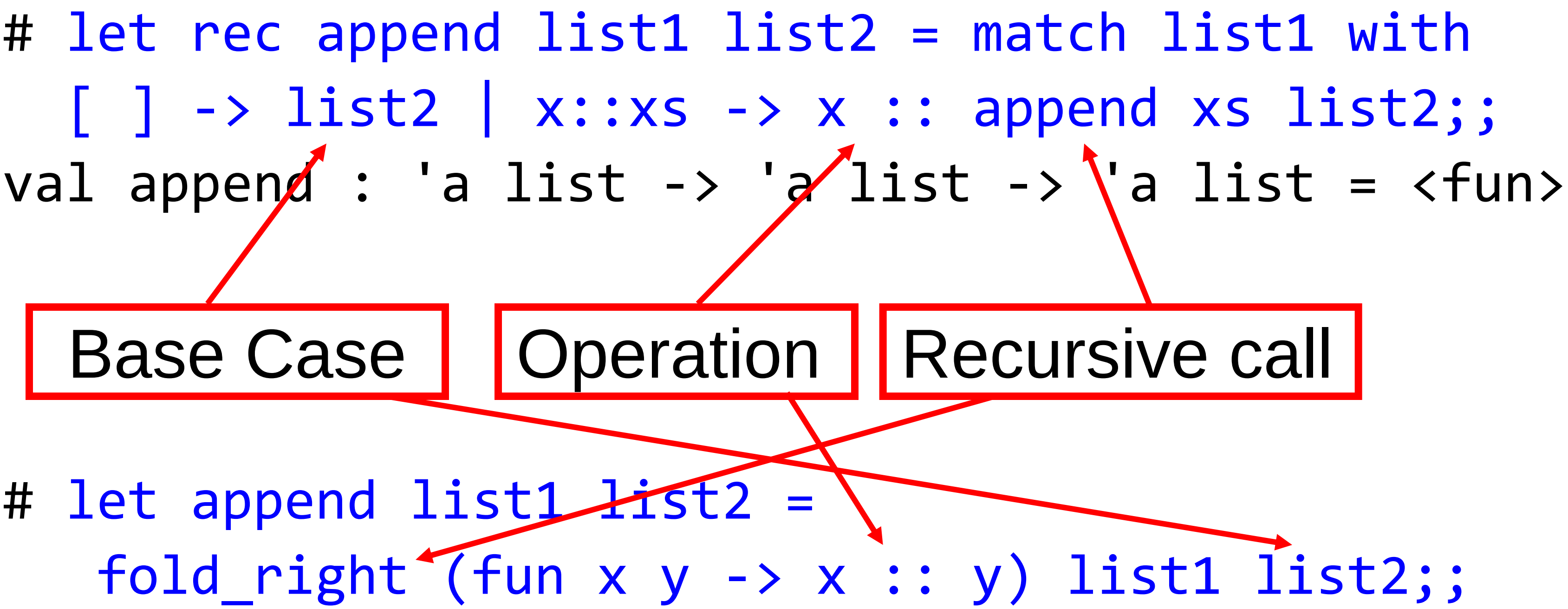
```
# let rec fold_right f list b =
  match list with
    [] -> b
  | (x :: xs) -> f x (fold_right f xs b);;

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

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
```

Encoding Recursion with Fold



```
# 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

Operation

Recursive call

```
# let append list1 list2 =
   fold_right (fun x y -> x :: y) list1 list2;;
val append : 'a list -> 'a list -> 'a list = <fun>
# append [1;2;3] [4;5;6];;
 - : int list = [1; 2; 3; 4; 5; 6]
```

Question

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

- How do you write length with fold_right, but no explicit recursion?

Question

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

- How do you write length with fold_right, but no explicit recursion?

```
let length list =
   List.fold_right (fun x -> fun n ->  n + 1)
                   list 0
```

Question

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

- How do you write length with fold_right, but no explicit recursion?

```
let length list =
   List.fold_right (fun x -> fun n ->  n + 1)
                   list 0
```

Can you write fold_right (or fold_left) with just map? How, or why not?

Iterating over lists

```
# 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 = ()
```

Can you do this?

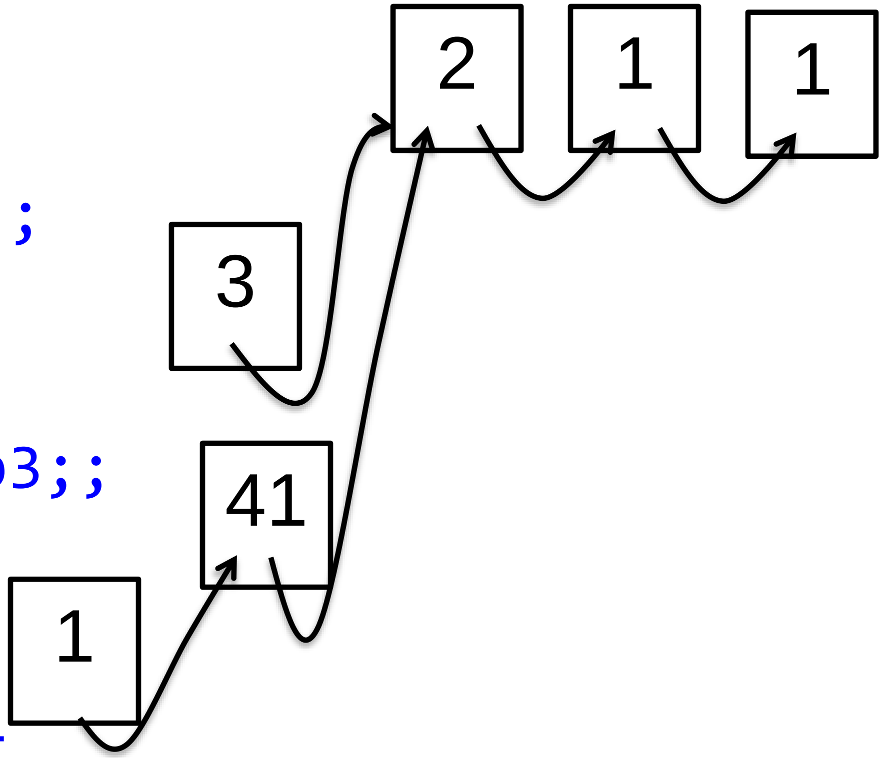
Recall:

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

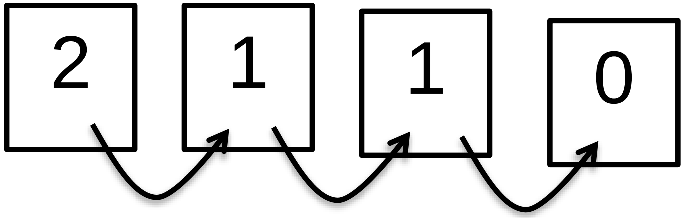
How can you implement map via fold_right or fold_left?

Back to Lists
(Data structures are immutable!)

```
# let fib3 = [2;1;1];;

# let fib4 = 3 :: fib3;;

# let fib41 = 41 :: fib3;;

# let fibI = 1 :: fib41
```

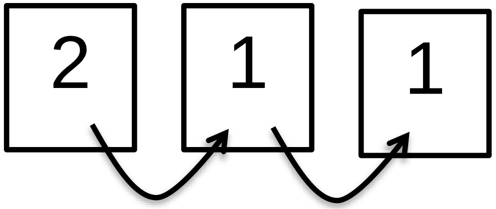


```
# let fib0 = fib3 @ [0];;
```



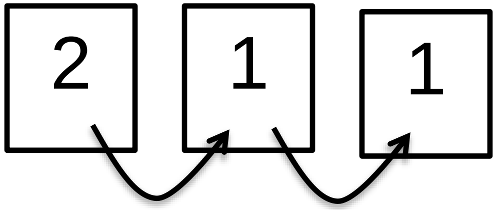
Data Structures are immutable

`mylist:`

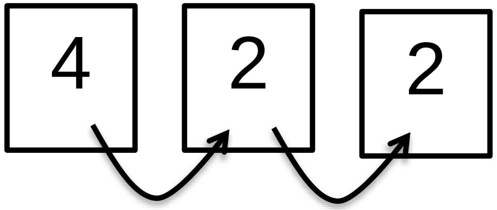


```
# let doubleList list =
    List.map (fun x -> 2 * x) list;;

# let res = doubleList mylist;;
```

`mylist:`



`res:`



Naïve Imperative Code Can Hinder Parallelism

Recall:

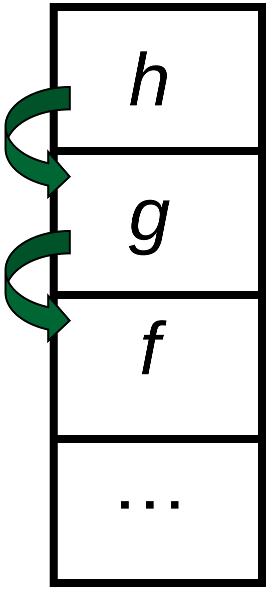
```
    int X[], Y[], a[], t, i;
    for i = 1 to N
S1:      t = a[i] + 2
S2:      Y[i] = t + 1
    end
```

Every iteration depends on the update of the index variable i

Moving on…

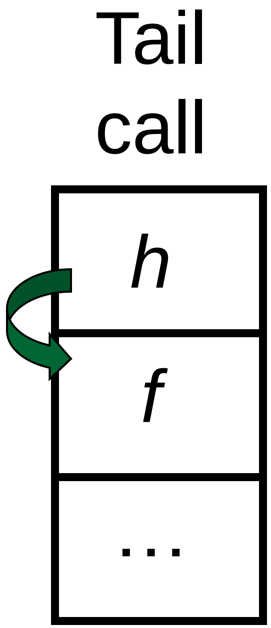
An Important Optimization

Normal call



- 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*)?
  - `let f x = (g x) + 1`
  - `let g x = h (x+1)`
  - `let h x = …`

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

Example of Tail Recursion

```
# let rec prod l =
    match l with [] -> 1
    | (x :: rem) -> x * prod rem;;
val prod : int list -> int = <fun>

# let prod list =
    let rec prod_aux l acc =
        match l with [] -> acc
        | (y :: rest) -> prod_aux rest (acc * y)
(* Uses associativity of multiplication *)
    in prod_aux list 1;;
 val prod : int list -> int = <fun>
```

Question

- How do you write length with tail recursion?

```
let length l =
```

Question

- How do you write length with tail recursion?

```
let length l =
    let rec length_aux list n =


in
```

Question

- How do you write length with tail recursion?

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

Question

- How do you write length with tail recursion?

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

Question

- How do you write length with tail recursion?

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

Question

- How do you write length with tail recursion?

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

Question

- How do you write length with tail recursion?

```
let length l =
    let rec length_aux list n =
    match list with [] -> n
     | (a :: bs) -> length_aux bs (n + 1)
in
```

Question

- How do you write length with tail recursion?

```
let length l =
    let rec length_aux list n =
    match list with [] -> n
     | (a :: bs) -> length_aux bs (n + 1)
in length_aux l 0
```

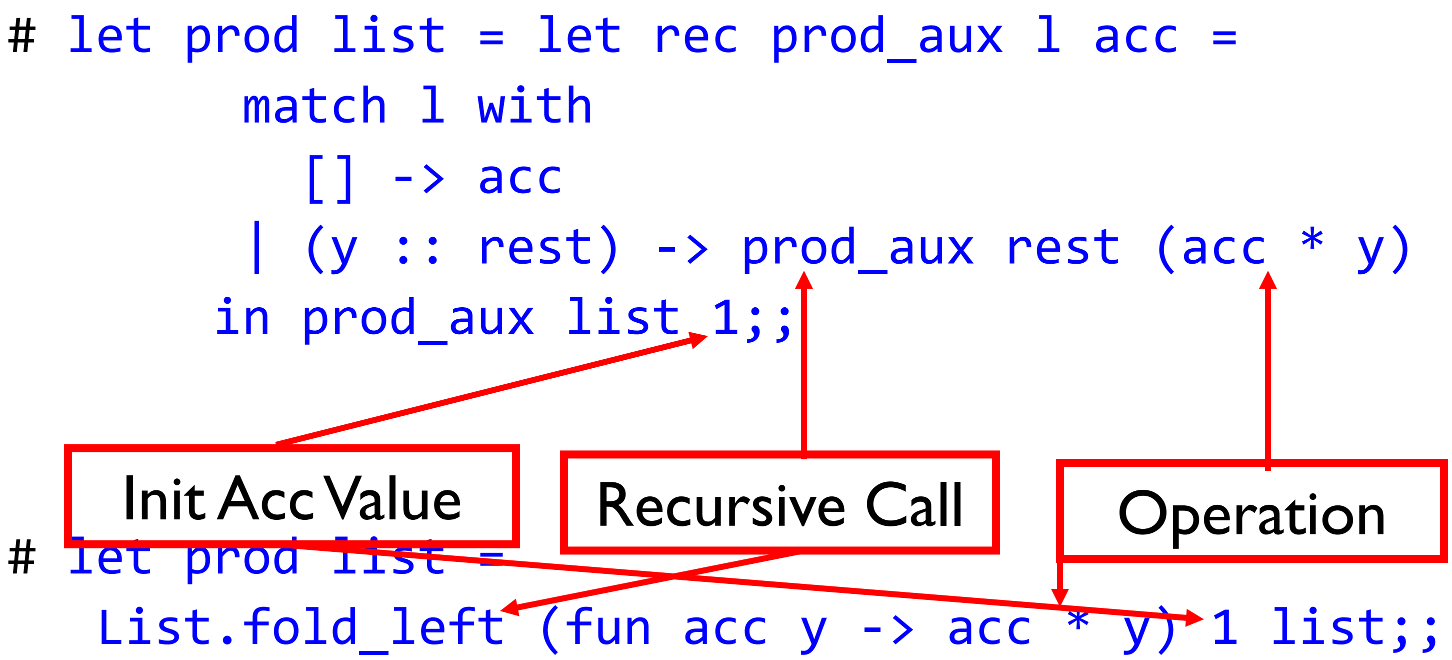
Your Turn

- Write a function odd_count_tr : int list -> int such that it returns the number of odd integers found in the input list. The function is required to use (only) tail recursion (no other form of recursion).

```
# let rec odd_count_tr l =
```

```
# odd_count_tr [1;2;3];;
- : int = 2
```

Encoding Tail Recursion with fold_left



```
# let prod list = let rec prod_aux l acc =
        match l with
          [] -> acc
        | (y :: rest) -> prod_aux rest (acc * y)
       in prod_aux list 1;;
```

```
# let prod list =
    List.fold_left (fun acc y -> acc * y) 1 list;;
```

```
# prod [4;5;6];;
 - : int =120
```

Question

```
let length l =
    let rec length_aux list n =
    match list with [] -> n
     | (a :: bs) -> length_aux bs (n + 1)
in length_aux l 0
```

- How do you write length with fold_left, but no explicit recursion?

Question

```
let length l =
    let rec length_aux list n =
    match list with [] -> n
     | (a :: bs) -> length_aux bs (n + 1)
in length_aux l 0
```

- How do you write length with fold_left, but no explicit recursion?

```
let length list =
 List.fold_left (fun n -> fun x -> n + 1)
                0 list
```

Folding

```
# 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

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

```
fold_left f 0 [1; 2; 3] = f (f (f 0 1) 2) 3
```

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

fold_right f’ [1; 2; 3] 0 = f’ x_1 (f’ x_2 (f 3 0))

Recall

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

- What is its running time?

Quadratic Time

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

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

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]

Tail Recursion - Example

```
# 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

- 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 - Tail Recursion

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

# 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

Example of Tail Recursion

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

# let app fs x =
    let rec app_aux fl acc =
        match fl with [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                                     (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>
```

Your turn now

Write a function

```
map_tail : (‘a -> ‘b) -> ‘a list -> ‘b list
```

that takes a function and a list of inputs and gives the result of applying the function on each argument, but in tail recursive form.

```
let make_app lst =
```

Continuation Passing Style

- A programming technique for all forms of "non-local" control flow:
  - non-local jumps
  - exceptions
  - general conversion of non-tail calls to tail calls
- Essentially it's a higher-order function version of GOTO

Continuations

- Idea: Use functions to represent the control flow of a program
- Method: Each procedure takes a function as an argument to which to pass its result; outer procedure "returns" no result
- Function receiving the result called a continuation
- Continuation acts as "accumulator" for work still to be done

Continuation Passing Style

- Writing procedures so that they take a continuation to which to give (pass) the result, and return no result, is called continuation passing style (CPS)

Example

- Simple reporting continuation:

```
# let report x = (print_int x;
                  print_newline( ) );;
val report : int -> unit = <fun>
```

- Simple function using a continuation:

```
# let plusk a b k = k (a + b)
val plusk : int -> int -> (int -> ’a) -> ’a
  = <fun>
# plusk 20 22 report;;
42
- : unit = ()
```

Example of Tail Recursion & CSP

```
# let app fs x =
    let rec app_aux fl acc=
        match fl with
          [] -> acc
        | (f :: rem_fs) -> app_aux rem_fs
                                    (fun z -> acc (f z))
    in app_aux fs (fun y -> y) x;;
val app : ('a -> 'a) list -> 'a -> 'a = <fun>

# let rec appk fl x k =
    match fl with
      [] -> k x
    | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));
hval appk : ('a -> 'a) list -> 'a -> ('a -> 'b) -> 'b
```

Example of Tail Recursion & CSP

```
# let rec appk fl x k =
    match fl with
      [] -> k x
    | (f :: rem_fs) -> appk rem_fs x (fun z -> k (f z));;

# appk [(fun x->x+1); (fun x -> x*5)] 2 (fun x->x);;
- : int = 11
```

Continuation Passing Style

- A compilation technique to implement non-local control flow, especially useful in interpreters.
- A formalization of non-local control flow in denotational semantics
- Possible intermediate state in compiling functional code

Optional: Matrix Multiply in Ocaml

Inputs:

- 1. matA – m x n matrix as row-major list of lists
- 2. matBT – transposed matrix (p x n before, n x p after transpose) as column-major list of lists

Exist implementations of map, fold_right, map2 (do them!)

```
let dotprod vec1 vec2 =    (* dot product of two vectors *)
  let prods = map2 ( *. ) vec1 vec2 in
  fold_right ( +. ) prods 0.0 ;;

let matmul matA matBT =    (* multiply A with transposed B *)
  map (fun row -> map (fun col -> dotprod row col) matBT) matA

let checkdim matA matBT = true / false ;;
(* For you: ensure columns and rows > 0 for both and also that
    colsA = rowsB  (because B is transposed) *)
```

Optional: Neural Network in Ocaml

```
 let inputs =                [[0.1; 0.2; -0.3];
  - matrix of NN inputs       [0.2; -0.1; 0.2] ];;
 let weightsT =              [[1.0; 0.1; -0.2];
  - transposed  matrix of     [-3.0; 1.1; -0.5];
  weights for all neurons     [-1.0; 0.1; 2.0] ];;


let relu x = if x > 0.0 then x else 0.0 ;;
let activation func matrix =
  map (fun row -> map func row) matrix ;;

(* fully connected layer *)
let fc1 = activation relu (matmul inputs weightsT) ;;

(* then we can chain multiple layers - each with own weights *)
let fc2 = activation relu (matmul fc1 weights2T) ;; (* etc. *)
let fc3 = activation relu (matmul fc3 weights3T) ;;
```