Programming Languages and Compilers (CS 421)

#5: Recursion, lists, forward/head rec, tail rec, maps

#6: Higher-order recursion, fold left/right, intro to CPS

Madhusudan Parthasarathy

http://courses.engr.illinois.edu/cs421

Based on slides by Elsa Gunter, which in turn is partly based on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Recursive Functions

```
# let rec factorial n =
    if n = 0 then 1 else n * factorial (n - 1);;
  val factorial : int -> int = <fun>
# factorial 5;;
-   : int = 120
```

"rec" keyword needed in Ocaml for recursive function declarations

Recursion Example

Compute n^2 recursively using:

$$n^2 = (2 * n - 1) + (n - 1)^2$$

```
# 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) ;;
```

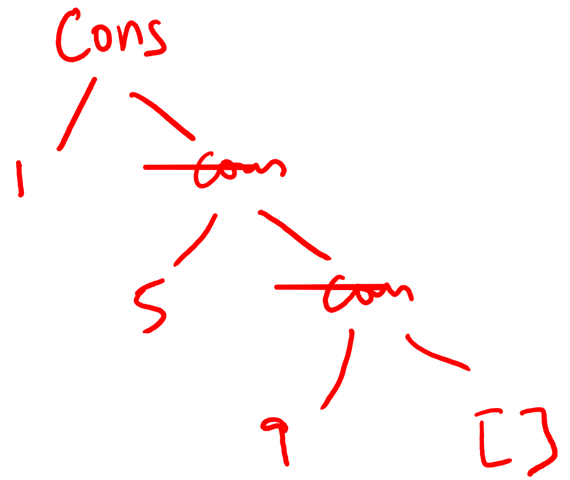
For termination:

- 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)

Lists

[1 ; 5 ; 9] = 1 :: (5 :: (9 :: [ ]))

= cons(1, cons(5, cons(9, [ ])))

- List can take one of two forms:
  - Empty list, written [ ]
  - Non-empty list, written x :: xs

    Cons(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
```

~~append~~(x , y)   (append x y)

```
# 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

x::x::t   l

_::t

```
# 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"]
```

append ((poor_rev xs) (x::[]))

Question: Length of list

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

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

Question: Length of list

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

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

Same Length

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

l_1	l_2	
[ ]	[ ]	T
[ ]	x::xs	F
~~x::xs~~	[ ]	F
x::xs	y::ys	
	cmp xs 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)
```

Structural Recursion

- "Everything is a tree"
- Lists as terms/trees; recursion on terms/trees
- Algebraic datatypes

Structural Recursion

[ ]

Cons

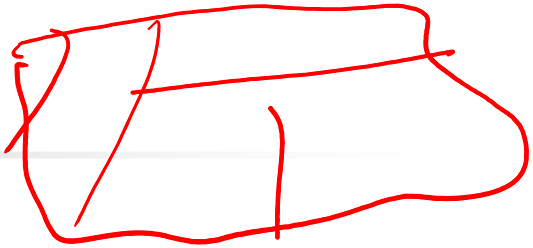
- Functions on recursive datatypes (e.g. lists) tend to be recursive
- Recursion over recursive datatypes generally by structural recursion
  - Recursive calls made to components of structure of the same recursive type
  - Base cases of recursive types stop the recursion of the function

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/head Recursion



- In Structural Recursion, split input into components and (eventually) recurse on components, and compute based on their results
- 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 all substructures has been worked on before 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>
```

```
| (x::xs) -> let r = double_up xs
             in   x :: x :: r
```

```
Cons (x, Cons (x, (double_up xs)))
```

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

Question

- How do you write length with forward recursion?

```
let rec length l =
```

Question

- How do you write length with forward recursion?

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

Question

- How do you write length with forward recursion?

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

Question

- How do you write length with forward recursion?

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

Your turn now

Try Problem 2 on ML2

Aggregation

Compute the product of the numbers in a list:

Version 1:

```
let rec prod l =
match l with [] -> 1
      | (h :: t) -> (h * (prod t))
```

Aggregation

Compute the product of the numbers in a list:

Version 1:

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

Aggregation

Compute the product of the numbers in a list:

Version 2:

```
let prod l = prod_aux l 1
let rec prod_aux l p =
  match l with [] -> p
    | (x::xs) -> (prod_aux xs x*p)
```

Aggregation

Compute the product of the numbers in a list:

Version 2:

```
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;;
 val prod : int list -> int = <fun>
```

Difference between the two versions

prod([5;4;9;11])

Version 1:

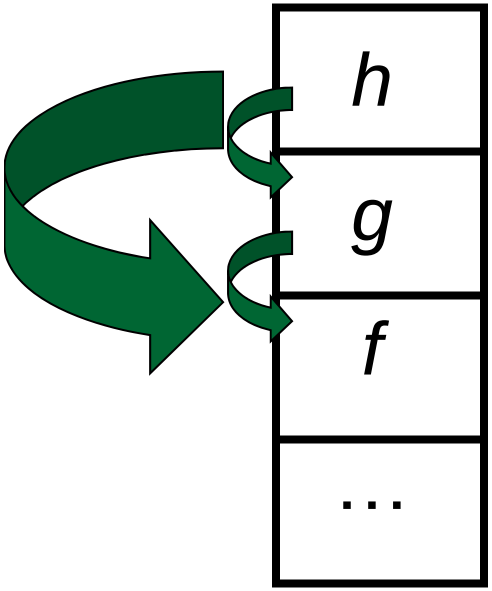
5*prod([4;9;11]=5*(4*prod([9;11]))
= 5*(4*(9*prod([11]))) = 5*(4*(9*(11*prod([]))))
= 5*(4*(9*(11*1)))

Version 2:

prod_aux([5;4;9;11], 1)
= prod_aux([4;9;11], 1*5)
= prod_aux([9;11], (1*5)*4)
= prod_aux([11], ((1*5)*4)*9)
= prod_aux([], (((1*5)*4)*9)*11) = (((1*5)*4)*9)*11

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*)?

- 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 (while) 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

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 [] -> n
     | (a :: bs) -> length_aux bs (n + 1)
in length_aux l 0
```

Your turn now

Try Problem 4 on MP2

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

- 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 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 l =

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;;
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
```

let sumlist list
= fold_~~right~~
(fun h -> fun r
-> h+r)
list
0

---

let prodlist list
= (fun h -> fun r
-> h * r)
list
1

Folding Functions over Lists

How are the following functions similar?

```
# 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 multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [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;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [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;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [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;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * multList xs;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24
```

Combining Operation

Folding Functions over Lists

How are the following functions similar?

```
# let rec sumList list = match list with
  [ ] -> 0 | x::xs -> x + Rec value ;;
val sumList : int list -> int = <fun>
# sumList [2;3;4];;
- : int = 9
# let rec multList list = match list with
  [ ] -> 1 | x::xs -> x * Rec value ;;
val multList : int list -> int = <fun>
# multList [2;3;4];;
- : int = 24R
```

Combining Operation

fold_right

~~let rec~~

fold_right f l b

= match l with [] → b

| (x::xs) → (f (x (fold_right f xs b))

Recursing over lists: fold_right


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 = ()
```

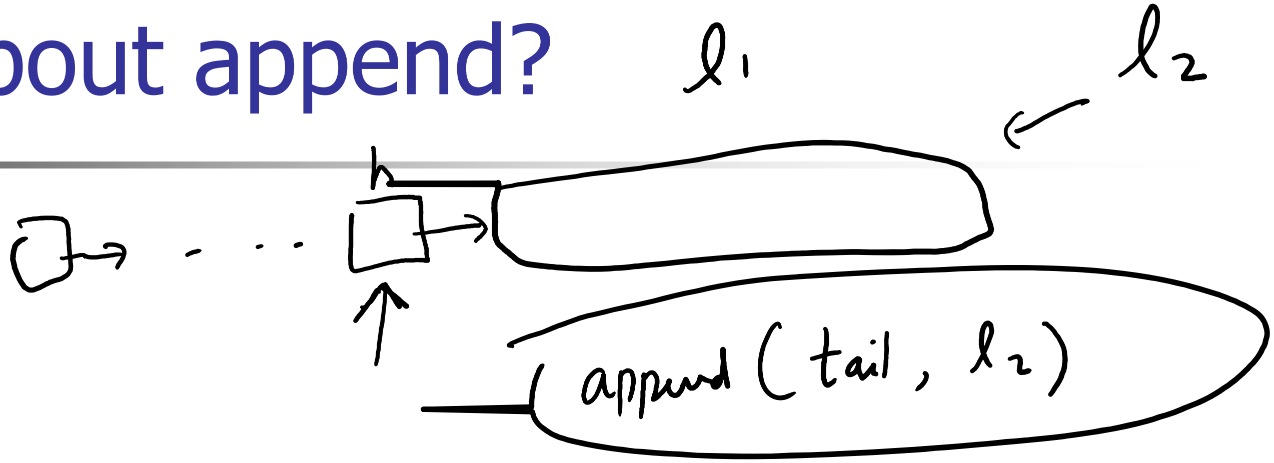
Folding Recursion

- 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
```

What about append?



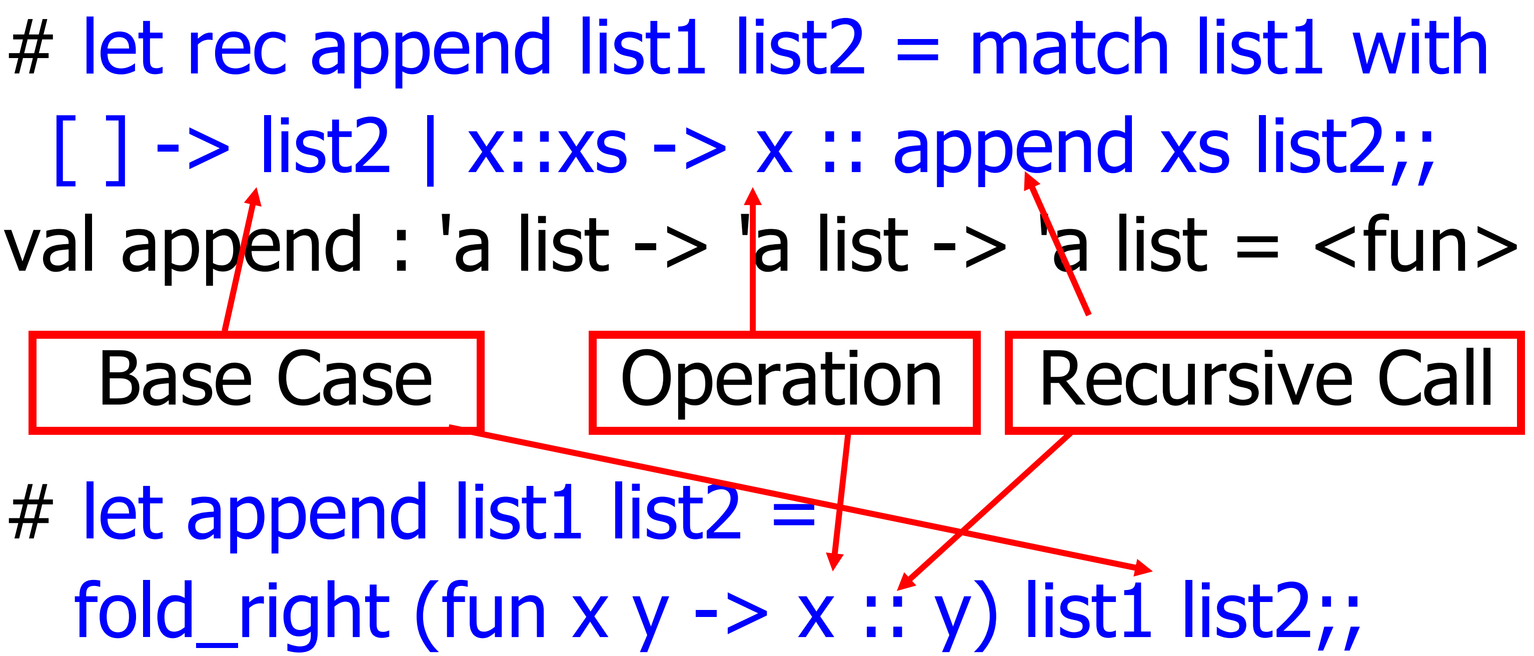
let

append l_1 l_2

$$\overset{h :: r}{}$$

= fold_right (fun h → fun r → h::r) l_1 l_2

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?

```
let length l = fold_right (fun h -> fun r ->
                                         1+r)
                          l
                          0
```

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

What about map?

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

let map f ~~list~~ =
fold_right (fun h -> fun r -> (f h) :: r)
list
[]

Map from Fold

```
# let map f list =
 fold_right (fun x -> fun y -> f x :: y) list [ ];;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
# map ((+)1) [1;2;3];;
- : int list = [2; 3; 4]
```

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

Iterating over lists: 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;;
```

f

```
let sum list =
 let rec sum_aux l acc =
  match l with [] -> acc
   | (y :: rest) -> sum_aux rest (acc + y)
 in sum_aux list 0;;
```

f'

initial value of acc

let rec

fold_left f a list

= match list with

[] → a

| y::rest →

fold_left f

(f a y)

rest

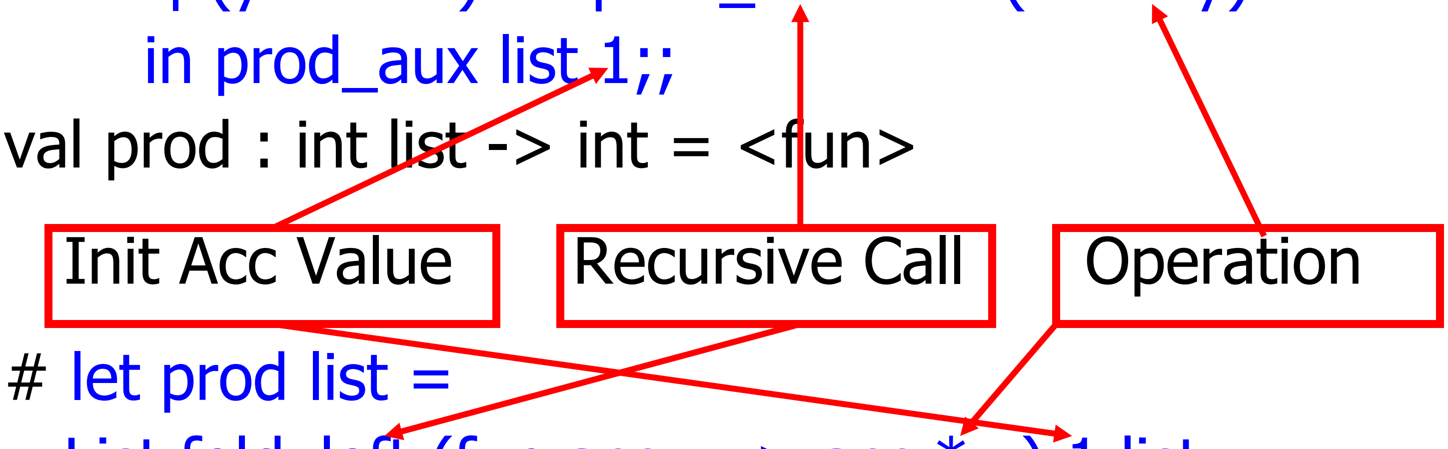
Iterating over lists: fold_left

```
# 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>

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

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;;
val prod : int list -> int = <fun>
```



```
# let prod list =
   List.fold_left (fun acc y -> acc * y) 1 list;;
val prod: int list -> int = <fun>
# 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?

let length l = fold_left
fun acc -> fun h -> (1+ acc)
0
l

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

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

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

Recall

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

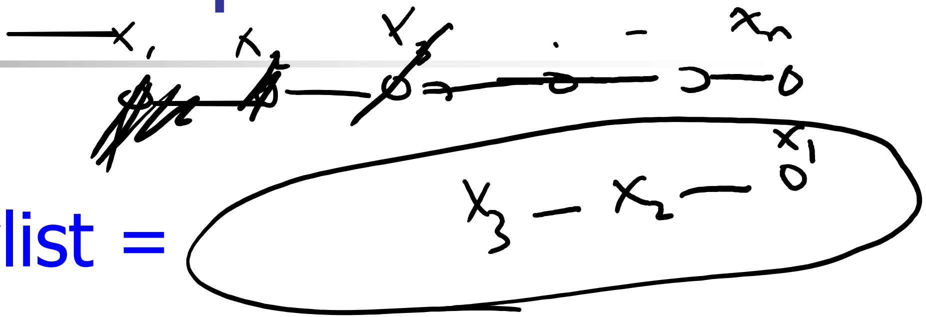
- What is its running time?

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>
```

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

- 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]

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 rev list =
-         fold_left
-          (fun acc -> fun h ->
                         h :: acc)
           [ ]
           list
```

Folding - Tail Recursion

```
# let rev list =
        fold_left
          (fun l -> fun x -> x :: l)     //comb op
         []              //accumulator cell
         list
```

```
/* Link list node */
struct Node
{   int data;     struct Node* next;  };

/* Function to reverse the linked list */
static void reverse(struct Node** head_ref)
{
   struct Node* prev   = NULL;
   struct Node* current = *head_ref;
   struct Node* next;
   while (current != NULL)
   {  next  = current->next;
      current->next = prev;
      prev = current;
      current = next;
   }
   *head_ref = prev;
}
```

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

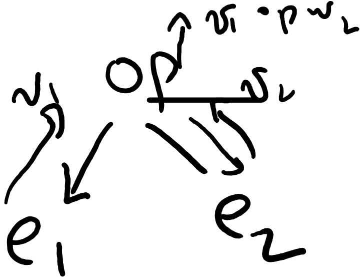
- Can replace recursion by fold_left in any tail primitive recursive definition

Recursion on trees: hard for tail recursion

((root) ( ) ( ))

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

- *Tail-recursion on acid*

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