# CMSC 330
# Organization of Programming Languages

## OCaml
## Higher Order Functions

# Anonymous Functions

- Recall code blocks in Ruby

```
(1..10).each { |x| print x }
```

  - Here, we can think of `{ |x| print x }` as a function

- We can do this (and more) in OCaml

# Anonymous Functions

- As with Ruby, passing around functions is common
  - So often we don't want to bother to give them names

- Use fun to make a function with no name

Parameter

Body
(in which parameter x
is bound)

```
fun x -> x + 3
```

```
(fun x -> x + 3) 5
```

```
= 8
```

# Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in
(fun z -> z-1) y
```

A. *Error*

B. `2`

C. `1`

D. `0`

# Quiz 1: What does this evaluate to?

```
let y = (fun x -> x+1) 2 in
(fun z -> z-1) y
```

A. *Error*

B. `2`

C. `1`

D. `0`

# Quiz 2: What is this expression's type ?

```
(fun x y -> x) 2 3
```

A. *Type error*

B. `int`

C. `int -> int -> int`

D. `'a -> 'b -> 'a`

# Quiz 2: What is this expression's type ?

```
(fun x y -> x) 2 3
```

*A. Type error*

**B. `int`**

C. `int -> int -> int`

D. `'a -> 'b -> 'a`

# Functions and Binding

- Functions are first-class, so you can bind them to other names as you like

```
let f x = x + 3;;
let g = f;;
g 5          = 8
```

- In fact, let for functions is a syntactic shorthand

```
let f x = body
```

↓ is semantically equivalent to

```
let f = fun x ->  body
```

# Example Shorthands

- `let next x = x + 1`
  - Short for `let next = fun x -> x + 1`

- `let plus x y = x + y`
  - Short for `let plus = fun x y -> x + y`

# Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
let h = fun y -> g (y+1) in
h 1
```

A. 0

B. 1

C. 2

*D. Error*

# Quiz 3: What does this evaluate to?

```
let f = fun x -> 0 in
let g = f in
let h = fun y -> g (y+1)
h 1
```

A. 0
B. 1
C. 2
D. *Error*

# Defining Functions Everywhere

```
let move l x =
  let left x = x - 1 in  (* locally defined fun *)
  let right x = x + 1 in (* locally defined fun *)
  if l then left x
  else      right x
;;

let move' l x = (* equivalent to the above *)
  if l then (fun y -> y - 1) x
  else      (fun y -> y + 1) x
```

# Pattern Matching With Fun

- match can be used within fun

```
(fun l -> match l with (h::_) -> h) [1; 2]    = 1
```

  - But use named functions for complicated matches

- May use standard pattern matching abbreviations, too

```
(fun (x, y) -> x+y) (1,2)    = 3
```

# Passing Functions as Arguments

In OCaml you can pass functions as arguments

```
let plus_three x = x + 3 (* int -> int *)

let twice f z = f (f z) (* ('a->'a) -> 'a -> 'a *)
```

**`f`**'s type

**`twice`**'s **`f`** parameter is a function

Calls the parameter function **`f`** (twice!)

```
twice plus_three 5   = 11
```

# `map`

# The Map Function

OCaml's **`map`** is a higher order function; like Ruby's **`collect`**

- **`map f l`** takes a function **`f`** and a list **`l`**, applies function **`f`** to each element of **`l`**, and returns a list of the results (preserving order)

```
map f [v1; v2; …; vn]
    = [f v1; f v2; …; f vn]
```

```
let add_one x = x + 1
let negate x = -x
map add_one [1; 2; 3]  = [2; 3; 4]
map negate [9; -5; 0]  = [-9; 5; 0]
```

# How can we implement Map?

```
let rec add1all l =
  match l with
    [] -> []
  | h::t ->
      (add_one h):: add1all t
```

```
let rec negall l =
  match l with
    [] -> []
  | h::t ->
      (neg h):: negall t
```

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

# Implementing map

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

- What is the type of **`map`**?

$$\underbrace{(\qquad\qquad\qquad)}_{\texttt{f}}\ \texttt{->}\ \underbrace{\qquad\qquad\qquad}_{\texttt{l}}\ \texttt{->}$$

# Implementing map

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

- What is the type of **map**?

$$\underbrace{\texttt{('a -> 'b)}}_{\texttt{f}}\ \texttt{->}\ \underbrace{\texttt{'a list}}_{\texttt{l}}\ \texttt{-> 'b list}$$

# Another Example

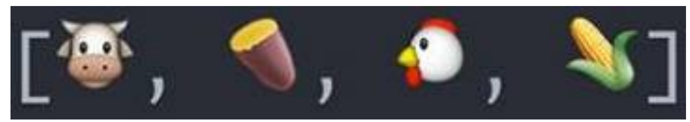
Apply a *list of functions* to *list of* ***`int`*****s**

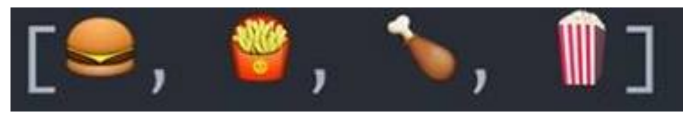
```
let neg x = -x;;
let add_one x = x+1;;
let double x = x + x;;
let fs = [neg; add_one; double];;
let lst = [1;2;3];;
```

```
map (fun f -> map f lst) fs =
[[-1; -2; -3]; [2; 3; 4]; [2; 4; 6]]
```

(`neg 1`) (`neg 2`) (`neg 3`) (`add_one 1`) … (`double 1`) …

# map, as a cartoon

**`map cook`**  =



**`map`** is included in the standard **`List`** module, i.e., as **`List.map`**

# Quiz 4: What does this evaluate to?

```
map (fun x -> x * 4) [1;2;3]
```

A. `[1.0; 2.0; 3.0]`
B. `[4.0; 8.0; 12.0]`
C. Error
D. `[4; 8; 12]`

# Quiz 4: What does this evaluate to?

```
map (fun x -> x * 4) [1;2;3]
```

A. `[1.0; 2.0; 3.0]`

B. `[4.0; 8.0; 12.0]`

C. Error

D. `[4; 8; 12]`

# Quiz 5: Which function to use?

```
map ??? [1; 0; 3] = [true; false; true]
```

```
A.  fun x -> true
B.  fun x -> x = 0
C.  fun x -> x != 0
D.  fun x -> x = (x != 0)
```

# Quiz 5: Which function to use?

```
map ??? [1; 0; 3] = [true; false; true]
```

A. `fun x -> true`

B. `fun x -> x = 0`

C. `fun x -> x != 0`

D. $\texttt{fun x -> } \underbrace{\texttt{x}}_{\texttt{int}} \texttt{ = } \underbrace{\texttt{(x != 0)}}_{\texttt{bool}}$

***Note type error!***

# `fold`

## `(and foldr)`

# Two Recursive Functions

Sum a list of ints

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

```
# sum [1;2;3;4];;
- : int = 10
```

Concatenate a list of strings

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

```
# concat ["a";"b";"c"];;
- : string = "abc"
```

# Notice Anything Similar?

Sum a list of ints

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

Concatenate a list of strings

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

# The foldr Function

Sum a list of ints

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

Concatenate a list of strings:

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

```
let rec foldr f a l =
  match l with
    [] -> a
  | h::t -> f h (foldr f a t)
```

```
let sum l = foldr (+) 0 l
```

```
let concat l = foldr (^) "" l
```

# So, What is foldr?

- foldr is a function that
  - takes a function of two arguments, a final value, and a list
  - processes the list by applying the function to the head and the recursive application of the function to the rest of the list, returning the final value for the empty list

```
foldr f v [v1; v2; …; vn] =
    f v1 (f v2 (…(f vn v)…))
```

```
so foldr add 0 [1;2;3;4] =
      add 1 (add 2 (add 3 (add 4 0))) = 10
```

# Foldr and the Standard Library

- **`List.fold_right`** in the standard library is **`foldr`**, but with the order of its last two parameters reversed, i.e.,

```
fold_right f [v1; v2; …; vn]  v =
    f v1 (f v2 (…(f vn v)…))
```

```
so fold_right add [1;2;3;4] 0 =
       add 1 (add 2 (add 3 (add 4 0))) = 10
```

# Fold (aka fold_left)

- The **`List`** module also defines **`fold_left`**
  - which we will just call **`fold`**

```
let rec fold f a l =
  match l with
    [] -> a
  | h::t -> fold f (f a h) t
```

Computes **`f`** on the *accumulator* **`a`** and the head **`h`**, then passes the result as the accumulator to the recursive call

  - Similar to **`foldr`**, but changes the order of operations

```
let rec foldr f a l =
  match l with
    [] -> a
  | h::t -> f h (foldr f a t)
```

# What does `fold` do?

```
let rec fold f a l =
  match l with
    [] -> a
  | h::t -> fold f (f a h) t
```

```
let add a x = a + x
fold add 0         [1; 2; 3] →
fold add (add 0 1) [2; 3] →
fold add 1         [2; 3] →
fold add (add 1 2) [3] →
fold add 3         [3] →
fold add (add 3 3) [] →
fold add 6         [] →
6
```

We just built the `sum` function!

# Fold (aka fold_left)

```
let rec fold f a l =
  match l with
    [] -> a
  | h::t -> fold f (f a h) t
```

- What does **`fold`** do?

- **fold** *f* *v* [*v1*; *v2*; …; *vn*]

= **fold** *f* (*f* *v* *v1*) [*v2*; …; *vn*]

= **fold** *f* (*f* (*f* *v* *v1*) *v2*) […; *vn*]

= …

= *f* (*f* (*f* (*f* *v* *v1*) *v2*) …) *vn*

  - e.g., `fold add 0 [1;2;3;4] =`

    `add (add (add (add 0 1) 2) 3) 4 = 10`

# Another Example

```
let rec fold f a l =
  match l with
    [] -> a
  | h::t -> fold f (f a h) t
```

```
let next a _ = a + 1
fold next 0             [2; 3; 4] →
fold next (next 0 2) [3; 4] →
fold next 1             [3; 4] →
fold next (next 1 3) [4] →
fold next 2             [4] →
fold next (next 2 4) [] →
fold next 3 [] →
3
```

We just built the `length` function!

# Using Fold to Build Reverse

```
let rec fold f a l =
  match l with
    [] -> a
  | h::t -> fold f (f a h) t
```

- Let's build the reverse function with fold!

```
let prepend a x = x::a
fold prepend [] [1; 2; 3; 4] →
fold prepend [1] [2; 3; 4] →
fold prepend [2; 1] [3; 4] →
fold prepend [3; 2; 1] [4] →
fold prepend [4; 3; 2; 1] [] →
[4; 3; 2; 1]
```

# Quiz 6: What does this evaluate to?

```
let f x y = if x > y then x else y in
fold f 0 [3;4;2]
```

A. 0

B. true

C. 2

D. 4

# Quiz 6: What does this evaluate to?

```
let f x y = if x > y then x else y in
fold f 0 [3;4;2]
```

A. `0`

B. `true`

C. `2`

D. `4`

# Quiz 7: What does this evaluate to?

```
fold (fun a y -> a-y) 0 [3;4;2]
```

A. -9

B. -1

C. [2;4;3]

D. 9

# Quiz 7: What does this evaluate to?

```
fold (fun a y -> a-y) 0 [3;4;2]
```

A. **-9**

B. -1

C. [2;4;3]

D. 9

# Type of fold_left, fold_right

```
let rec fold_left f a l =
  match l with
    [] -> a
  | h::t -> fold_left f (f a h) t
```

$$\underbrace{(\qquad\qquad\qquad)}_{\texttt{f}} \texttt{->} \underbrace{\qquad}_{\texttt{a}} \texttt{->} \underbrace{\qquad\qquad}_{\texttt{l}} \texttt{->}$$

# Type of fold_left, fold_right

```
let rec fold_left f a l =
  match l with
    [] -> a
  | h::t -> fold_left f (f a h) t
```

$$\underbrace{\texttt{('a -> 'b -> 'a)}}_{\texttt{f}}\ \texttt{->}\ \underbrace{\texttt{'a}}_{\texttt{a}}\ \texttt{->}\ \underbrace{\texttt{'b list}}_{\texttt{l}}\ \texttt{->}\ \texttt{'a}$$

# Type of fold_left, fold_right

```
let rec fold_left f a l =
  match l with
    [] -> a
  | h::t -> fold_left f (f a h) t
```

$$\underbrace{(\texttt{'a -> 'b -> 'a})}_{\texttt{f}}\ \texttt{->}\ \underbrace{\texttt{'a}}_{\texttt{a}}\ \texttt{->}\ \underbrace{\texttt{'b list}}_{\texttt{l}}\ \texttt{->}\ \texttt{'a}$$

```
let rec fold_right f l a =
  match l with
    [] -> a
  | h::t -> f h (fold_right f t a)
```

$$\underbrace{(\texttt{'b -> 'a -> 'a})}_{\texttt{f}}\ \texttt{->}\ \underbrace{\texttt{'b list}}_{\texttt{l}}\ \texttt{->}\ \underbrace{\texttt{'a}}_{\texttt{a}}\ \texttt{->}\ \texttt{'a}$$

# Summary: Left-to-right vs. right-to-left

**fold_left** *f* *v* [*v1*; *v2*; …; *vn*] =
  *f* (*f* (*f* (*f* *v* *v1*) *v2*) …) *vn*

**fold_right** *f* [*v1*; *v2*; …; *vn*] *v* =
  *f* *v1* (*f* *v2* (… (*f* *vn* *v*) …))

```
fold_left (fun x y -> x - y) 0 [1;2;3] = -6
```

since ((0-1)-2)-3) = -6

```
fold_right [1;2;3] (fun x y -> x - y) 0 = 2
```

since 1-(2-(3-0)) = 2

# When to use one or the other?

- Many problems lend themselves to `fold_right`
- But it does present a performance disadvantage
  - The recursion builds of a deep stack: One stack frame for each recursive call of fold_right
- An optimization called tail recursion permits optimizing `fold_left` so that it uses no stack at all
  - We will see how this works in a later lecture!

# Combining map and fold

- Idea: map a list to another list, and then fold over it to compute the final result
  - Basis of the famous "map/reduce" framework from Google, since these operations can be parallelized

```
let countone l =
  fold (fun a h -> if h=1 then a+1 else a) 0 l

let countones ss =
  let counts = map countone ss in
  fold (fun a c -> a+c) 0 counts

countones [[1;0;1]; [0;0]; [1;1]] = 4
countones [[1;0]; []; [0;0]; [1]] = 2
```

# `fold` & `map`

More examples, practice

# Map Example 1: Permute a list

```
let permute lst =
  let rec rm x l = List.filter ((<>) x) l
  and insertToPermute lst x =
    let t = rm x lst in
    List.map ((fun a b->a::b) x )(permuteall t)
  and permuteall lst =
    match lst with
    |[]->[]
    |[x]->[[x]]
    |_->List.flatten(List.map (insertToPermute lst) lst)
  in permuteall lst
;;
```

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

# Map Example 2: Power Set

```
let populate a b =
  if b=[] then [[a]]
  else  let t = List.map (fun x->a::x) b in
        [a]::t@b
;;
```

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

```
let powerset lst = List.fold_right populate lst []
;;
```

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

# Fold Example 1: Product of an int list

```
let mul x y = x * y;;

let lst = [1; 2; 3; 4; 5];;

fold mul 1 lst
- : int = 120
```

Wrong accumulator

```
fold mul 0 lst;;
- : int = 0
```

# Fold Example 2: Count elements of a list satisfying a condition

```
let countif p l =
fold (fun counter element -> if p element then counter+1
                             else counter) 0 l ;;


countif (fun x -> x > 0) [30;-1;45;100;0];;

- : int = 3
```

# Fold Example 3: Collect even numbers in the list

```
let f acc y = if (y mod 2) = 0 then y::acc
              else acc;;

fold f [] [1;2;3;4;5;6];;

- : int list = [6; 4; 2]
```

Reversed

# Fold Example 4: Inner Product

first compute list of pair-wise products, then sum up

```
[x1;x2;x3]∗[y1;y2;y3] = x1∗y1 + x2∗y2 + x3∗y3
```

```
let rec map2 f a b =
       match (a,b) with
       |([],[])->([])
       |(h1::t1,h2::t2)->(f h1 h2):: (map2 f t1 t2)
       |_->invalid_arg "map2";;

let product v1 v2 =
      fold (+) 0 (map2 ( * ) v1 v2);;
# val product : int list -> int list -> int = <fun>
product [2;4;6] [1;3;5];;
#- : int = 44
```

# Fold Example 5: Find the maximum from a list

```
let maxList lst =
       match lst with
        []->failwith "empty list"
       |h::t-> fold max h t ;;

maxList [3;10;5];;
- : int = 10
```

```
(*
maxList [3;10;5]
fold max 3 [10:5]
fold max (max 3 10) [5]
fold max (max 10 5) []
fold max 10 []
10   *)
```

# Quiz: Sum of sublists

Given a list of int lists, compute the sum of each int list, and return them as list.

For example:

```
sumList [[1;2;3];[4];[5;6;7]]
- : int list = [6; 4; 18]
```

# Solution: Sum of sublists

```
let sumList  = map (fold (+) 0 );;

sumList [[1;2;3];[4;5;6];[10]];;
- : int list = [6; 15; 10]
```

# Quiz: Maximum contiguous sublist

Given an int list, find the contiguous sublist, which has the largest sum and return its sum.

Example:

Input: [-2,1,-3,**4,-1,2,1**,-5,4]

Output: 6

Explanation: [4,-1,2,1] has the largest sum = 6

# Quiz: Maximum contiguous sublist

```
let f (m, acc) h =
   let m = max m (acc + h) in
   let  x = if acc < 0 then 0 else acc in
   (m, x+h)
;;
let submax  lst = let (max_so_far, max_current) =
           fold f (0,0) lst in
           max_so_far
;;

 submax [-2; 1; -3; 4; -1; 2; 1; -5; 4];;
- : int = 6
```