CMSC 330
Organization of Programming Languages

OCaml

Higher Order Functions

Map & Fold

Passing Functions as Arguments

You can pass functions as arguments

```
let plus3 x = x + 3 (* int -> int *)

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


        twice plus3 5  = 11
```

The Map Function

`map` is a higher order function

```
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`**?

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

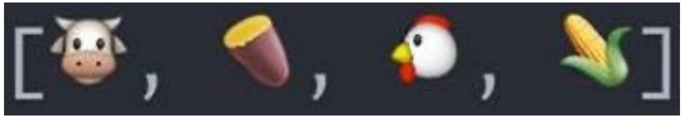
Implementing map

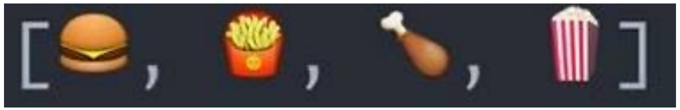
```
let rec map f l =
  match l with
    [] -> []
  | h::t -> (f h)::(map f t)
```

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

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

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`

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 fold Function

Sum a list of ints

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

Concatenate a list of strings:

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

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

```
let sum l = fold (+) 0 lst
let concat l = fold (^) "" lst
```

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!

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]
```

List.fold_left

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

- 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

List.fold_right

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

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

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

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
```

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

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
```

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

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

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

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!

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
```

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: 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
*)
```

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
```

Sum of sublists

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

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

For example:

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