CMSC 330
Organization of Programming Languages

OCaml
Higher Order Functions
Map & Fold
Passing Functions as Arguments

You can pass functions as arguments

\[
\text{let } \text{plus3 } x = x + 3 \quad (* \text{ int } \rightarrow \text{ int } *)
\]

\[
\text{let } \text{twice } f z = f (f z) \\
(* ('a->'a) \rightarrow 'a \rightarrow 'a *)
\]

\[
\text{twice plus3 5 } = 11
\]
The Map Function

map is a higher order function

\[
\text{map } f \ [v_1;\ v_2;\ \ldots;\ v_n]\\
= \ [f\ v_1;\ f\ v_2;\ \ldots;\ f\ v_n]
\]

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?

```ocaml
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

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

- What is the type of map?
Implementing map

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

- What is the type of `map`?

```ocaml
('a -> 'b) -> 'a list -> 'b list
```

- `f` and `l`
map, as a cartoon

\[ \text{map cook} = \left[ \begin{array}{ccccc} \text{cow} & \text{hotdog} & \text{chicken} & \text{corn} \\ \text{burger} & \text{fries} & \text{chicken} & \text{popcorn} \end{array} \right] \]

map is included in the standard List module, i.e., as List.map
Quiz 4: What does this evaluate to?

\[
\text{map } (\text{fun } x \rightarrow x \times 4) \ [1;2;3]
\]

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?

\[ \text{map (fun x -> x * 4) [1;2;3]} \]

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?

$$\text{map } ??? \ [1; 0; 3] = [true; false; true]$$

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. fun x -> x = (x != 0)

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 \texttt{fold} do?

\begin{verbatim}
let rec fold f a l =
  match l with
  [] -> a
| h::t -> fold f (f a h) t
\end{verbatim}

\texttt{let add a x = a + x}

\begin{align*}
\text{fold add 0} & \quad [1; 2; 3] \rightarrow \\
\text{fold add (add 0 1)} & \quad [2; 3] \rightarrow \\
\text{fold add 1} & \quad [2; 3] \rightarrow \\
\text{fold add (add 1 2)} & \quad [3] \rightarrow \\
\text{fold add 3} & \quad [3] \rightarrow \\
\text{fold add (add 3 3)} & \quad [] \rightarrow \\
\text{fold add 6} & \quad [] \rightarrow \\
\end{align*}

We just built the \texttt{sum} function!
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

\[
\text{fold}_\text{right} \ f \ [v_1; \ v_2; \ \ldots; \ v_n] \ v = \\
f \ v_1 \ (f \ v_2 \ (\ldots (f \ v_n \ v)\ldots))
\]

\[
\text{fold}_\text{right} \ \text{add} \ [1;2;3;4] \ 0 = \\
\text{add} \ 1 \ (\text{add} \ 2 \ (\text{add} \ 3 \ (\text{add} \ 4 \ 0))) = 10
\]

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

('a -> 'b -> 'a) -> 'a -> 'b list -> '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

('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
f a l

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

('b -> 'a -> 'a) -> 'b list -> 'a -> 'a
f l a
```
Summary: Left-to-right vs. right-to-left

\[
\text{fold\_left } f \ v \ [v1; v2; \ldots; vn] = \\
f (f (f (f \ v \ v1) \ v2) \ldots) \ vn
\]

\[
\text{fold\_right } f \ [v1; v2; \ldots; vn] \ v = \\
f \ v1 \ (f \ v2 (\ldots (f \ vn \ v) \ldots))
\]

\[
\text{fold\_left } (\text{fun } x \ y \to x - y) \ 0 \ [1;2;3] = -6 \\
\text{since } ((0-1)-2)-3) = -6
\]

\[
\text{fold\_right } [1;2;3] \ (\text{fun } x \ y \to x - y) \ 0 = 2 \\
\text{since } 1-(2-(3-0)) = 2
\]
When to use one or the other?

- Many problems lend themselves to `fold_right`
- But it does present a performance disadvantage
  - The recursion builds up 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

```ocaml
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]
Fold Example 4: Find the maximum from a list

```ocaml
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

```ocaml
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.

\[
\text{let sumList} = \text{map} \ (\text{fold} \ (+) \ 0) ;;
\]

For example:

\[
\text{sumList} \ [[1;2;3];[4];[5;6;7]]
\]
\[
- : \text{int list} = [6; 4; 18]
\]