CMSC 330:
Organization of Programming Languages

Map & Fold

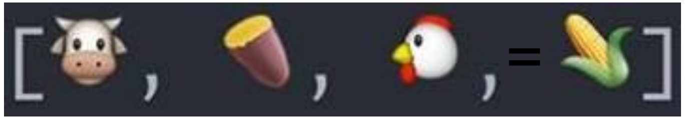
Spring 2026

The Map Function

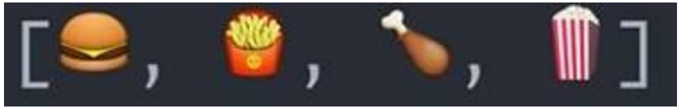
- `map` is a higher order function

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

map cook



[🐮, 🍠, 🐔, = 🌽]



[🍔, 🍟, 🍗, 🍿]

Implementing 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**?

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

Implementing map

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

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

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

Quiz: 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: 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]`

Fold

- Takes a list and collapses it into a single **value** by repeatedly applying a function.

```
    fold_left f init [x1; x2; x3]
```

Means

```
    f (f (f init x1) x2) x3
```

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!

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 l a=
  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
```

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
```

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

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
```

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

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!