CSE216
Foundations of Computer Science

Instructor: Zhoulai Fu

State University of New York, Korea

Many slides taken from Cornell's CS3110. Thanks!
https://www.cs.cornell.edu/courses/cs3110/2014fa/lecture_notes.php

Higher-order functions

- Functions are values

- Can use them **anywhere** we use values
  - Functions can **take** functions as arguments
  - Functions can **return** functions as results

    …functions can be *higher-order*

Map

```
map shirt_color [
```



```
]
```

```
= [gold, blue, red]
```

Map Implementation

```
let rec map f xs  =
    match xs with
       [] -> []
    | x::xs' -> (f x)::(map f xs')
```

```
map :  ('a -> 'b) -> 'a list -> 'b list
```

Implemented in List.map;

can be used in many data structures like queue, stack

Exercise

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = map is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false

Exercise

- Write a function **square_list** to convert a list of integers to a list of their squares.
    - 1. First, do not use **List.map**
    - 2. Then use **List.map**

Example:

Input: [1; 2; 3; 4]

Output: [1; 4; 9; 16]

Filter

```
filter is_vulcan [
```

```
]
```

```
= [
```

```
]
```

Filter (2)

```
let filter f xs =
    match xs with
       [] -> []
    | x::xs' -> if f x
                then x::(filter f xs')
                else filter f xs'
```

```
filter :  ('a -> bool) -> 'a list -> 'a list
```

Filter is also HUGE

- In library: **`List.filter`**

Exercise

What is value of lst after this code?

```
let is_even x = (x mod 2 = 0)
let lst = filter is_even [1;2;3;4]
```

A. [1;2;3;4]
B. [2;4]
C. [false; true; false; true]
D. false

Exercise

- Write a function **positive_list** to get from a list of integers to a list of its positive integers.
    - 1. First, do not use **List.map**
    - 2. Then use **List.map**

Example:

Input: [1; 2; -3; 4]

Output: [1; 2; 4]

Iterators

- Map and filter are *iterators*
  - Not built-in to the language, an idiom

- Benefit of iterators: separate recursive traversal from data processing
  - Can reuse same traversal for different data processing
  - leads to modular, maintainable, beautiful code!

- So far: iterators that change or omit data
  - what about combining data?
  - e.g., sum all elements of list

Folding v1.0

Idea: *stick an operator between every element of list*

folding `[1;2;3]` with `(+)`

becomes

1+2+3

-->

6

But list could have 1 element, so need an initial value

Folding v2.0

folding `[1;2;3]` with `0` and `(+)`

becomes

`0+1+2+3`

`-->`

`6`

Or list could be empty; just return initial value

folding **[ ]** with **0** and **(+)**

becomes

0

Question

What should the result of folding `[1;2;3;4]` with `1` and `(  *  )` be?

A. 1

B. 24

C. 10

D. 0

Implementation details

iterate left-to-right or right-to-left?

folding `[1;2;3]` with `0` and `(+)`

left to right becomes: ((0+1)+2)+3

right to left becomes: 1+(2+(3+0))

Both evaluate to 6; does it matter?

Yes: not all operators are associative, e.g. subtraction, division, exponentiation, …

Fold in Ocaml

Two versions in OCaml library:

```
List.fold_left
: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

```
List.fold_right
: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

fold_left

```
let rec fold_left f acc xs =
    match xs with
       []      -> acc
     | x::xs'  -> fold_left f (f acc x) xs'
```

Accumulates an answer by

- repeatedly applying **`f`** to “answer so far”,
- starting with initial value **`acc`**,
- folding “from the left”

`fold_left f acc [a;b;c]`
computes
`f (f (f acc a) b) c`

Google's Map-Reduce

- Fold has many synonyms/cousins in various functional languages, including **`scan`** and **`reduce`**
- Google organizes large-scale data-parallel computations with Map-Reduce
  - open source implementation by Apache called Hadoop

"[Google's Map-Reduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages. We realized that most of our computations involved applying a map operation to each logical record in our input in order to compute a set of intermediate key/value pairs, and then applying a reduce operation to all the values that shared the same key in order to combine the derived data appropriately."
[Dean and Ghemawat, 2008]

Fold is general

Implement so many other functions with fold!

```
let rev xs = fold_left (fun xs x -> x::xs) [] xs
let length xs = fold_left (fun a _ -> a+1) 0 xs
let map f xs = fold_right
   (fun x a -> (f x)::a) xs []
let filter f xs = fold_right
   (fun x a -> if f x then x::a else a) xs []
```

Exercise

- List.fold_left (+) 0 [1;2;3;4]

- List.fold_left (*) 1 [4;6;8]

Exercise

- List.fold_left (fun xs x -> x:: xs) [] [1;2;3;4]

Exercise

- List.fold_left (fun a _ -> a+1) 0 [1;2;3;4]