Higher-order Programming

Prof. Clarkson
Fall 2018

Today’s music: Selections from the soundtrack to 2001: A Space Odyssey
Attendance question

What do you think of A1?

A. It's a mystery.
B. It's a puzzle.
C. It's a riddle.
D. It's a conundrum.
E. It's an enigma.
Coding Standards Rubric

• **Meets Expectations** (0 points) is the norm

• **Needs Improvement** (-1 points) means you have room to improve and your TAs would be happy to help

• **Exceeds Expectations** (1 points) is rare and means you truly went beyond the call of duty
Review

Previously in 3110:
• Lots of language features

Today:
• No new language features
• New idioms and library functions:
  Map, fold, and other higher-order functions
Review: Functions are values

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

...so functions are higher-order
HIGHER-ORDER FUNCTIONS
TWO MONUMENTAL HIGHER-ORDER FUNCTIONS
map

fold

Sibling: reduce
MapReduce

“[Google’s MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages.”

[Dean and Ghemawat, 2008]
transform list elements

map

fold
Map

map (fun x -> shirt_color(x)) [ ]
Map

map (fun x -> shirt_color(x)) [ ]

= [gold; blue; red]
Map

*bad style!*

\[
\text{map} \left( \text{fun } x \rightarrow \text{shirt\_color}(x) \right) \left[ \text{[images]} \right] = \left[ \text{gold}; \text{ blue}; \text{ red} \right]
\]
Map

\[
\text{map shirt_color [ [ ] ]}
\]

= [gold; blue; red]
What is value of `lst` after this code?

```-scala
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`
TRANSFORMING ELEMENTS
**Map**

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

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

Factor out recurring code patterns.
Don't duplicate them.
map

fold

combine list elements
COMBINING ELEMENTS
Combining elements

```
let rec combine init op = function
  | [] -> init
  | h :: t ->
    op h (combine init op t)
```

combining elements, using `init` and `op`, is the essential idea behind library functions known as `fold`
**List.fold_right**

```
List.fold_right f [a;b;c] init
```

computes
```
f a (f b (f c init))
```

**Accumulates** an answer by

- repeatedly applying `f` to an element of list and “answer so far”
- folding in list elements “from the right”
List.fold_left

List.fold_left \( f \) \( \text{init} \) \([a;b;c]\)
computes
\( f \ (f \ (f \ \text{init} \ a) \ b) \ c \)

Accumulates an answer by
• repeatedly applying \( f \) to "answer so far" and an element of list
• folding in list elements “from the left”
Behold the power of 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 []
Difference 1: Left vs. right

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

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

In general, does left vs. right matter?
Question: A. Yes  B. No
Difference 2: Tail recursion

Which of these is tail recursive?

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

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

A. neither
B. fold_left
C. fold_right
D. both
Upcoming events

• [soon] A2-A4 teams announced

This is monumental.

THIS IS 3110