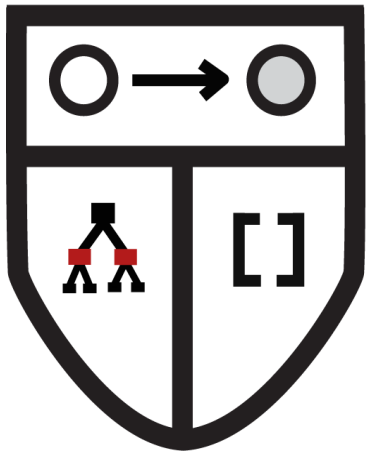


CS 3110

Higher-order Programming

Nate Foster
(Guest Lecture: Dexter Kozen)
Spring 2020

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

Demo

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]

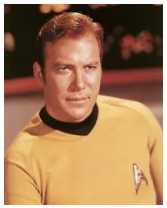
transform list elements

map

fold

Map

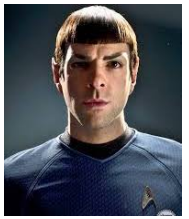
```
map (fun x -> shirt_color(x)) [
```



```
]
```

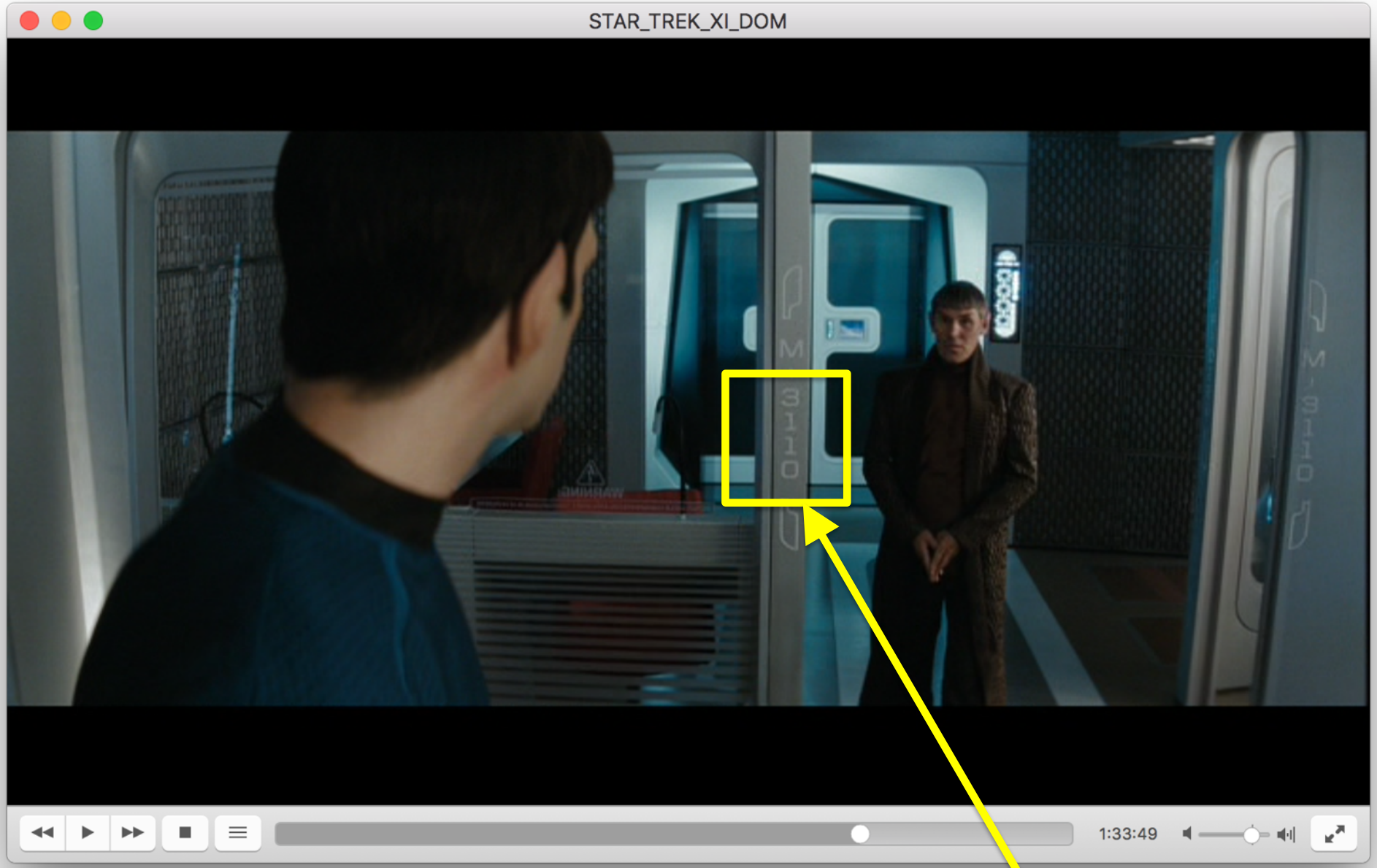
Map

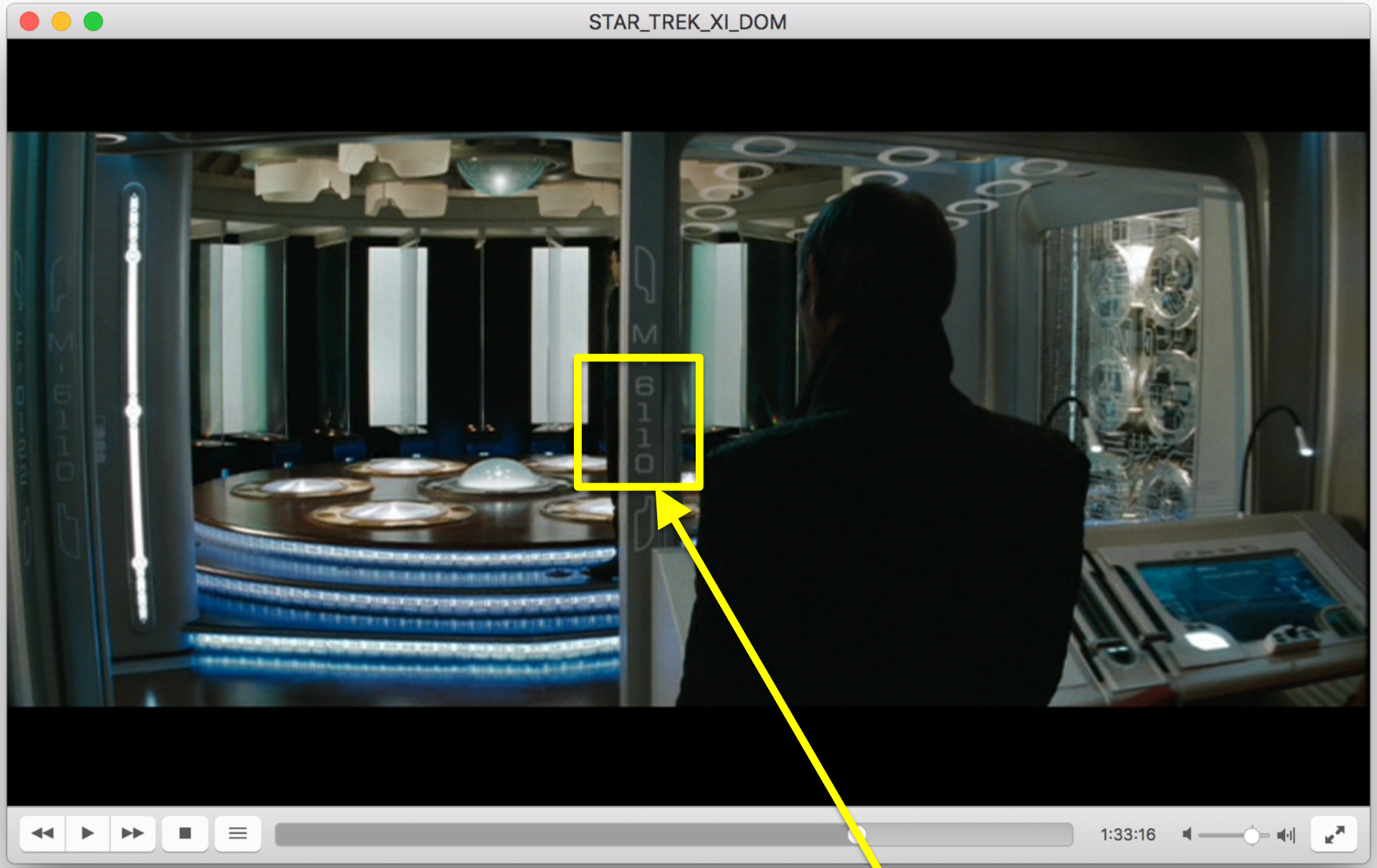
```
map (fun x -> shirt_color(x)) [
```


```
]
```

```
= [gold; blue; red]
```

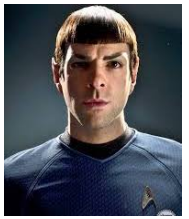

STAR_TREK_XI_DOM
1:33:49


STAR_TREK_XI_DOM
M 6110
1:33:16

Map

bad style!

```
map (fun x -> shirt_color(x))
```

[  ]

```
= [gold; blue; red]
```

Map

```
map shirt_color [
```



```
]
```

```
= [gold; blue; red]
```

TRANSFORMING ELEMENTS

Demo

Map

```
let rec map f = function
    | [] -> []
    | h :: t -> f h :: map f t
```

```
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

Demo

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"

Demo

`List.fold_left`

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

computes

```
f (f (f 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”

Demo

Left vs. right

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

left to right: ((0+1)+2)+3 = 6
right to left: 1+(2+(3+0)) = 6

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

left to right: ((0-1)-2)-3 = -6
right to left: 1-(2-(3-0)) = 2

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 []
```

Upcoming events

- [last night] A0 was due
- [Today] A1 out
- [Monday] R2 due

This is monumental.

THIS IS 3110