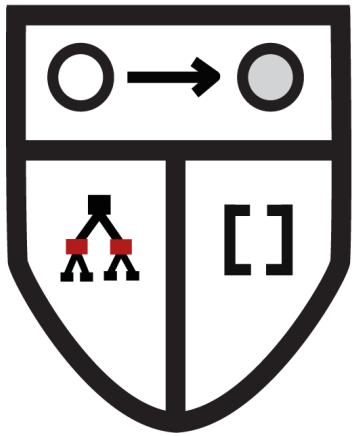


CS 3110

Higher-order Programming

Nate Foster

Spring 2019

Today's music: *Higher Ground* Stevie Wonder

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

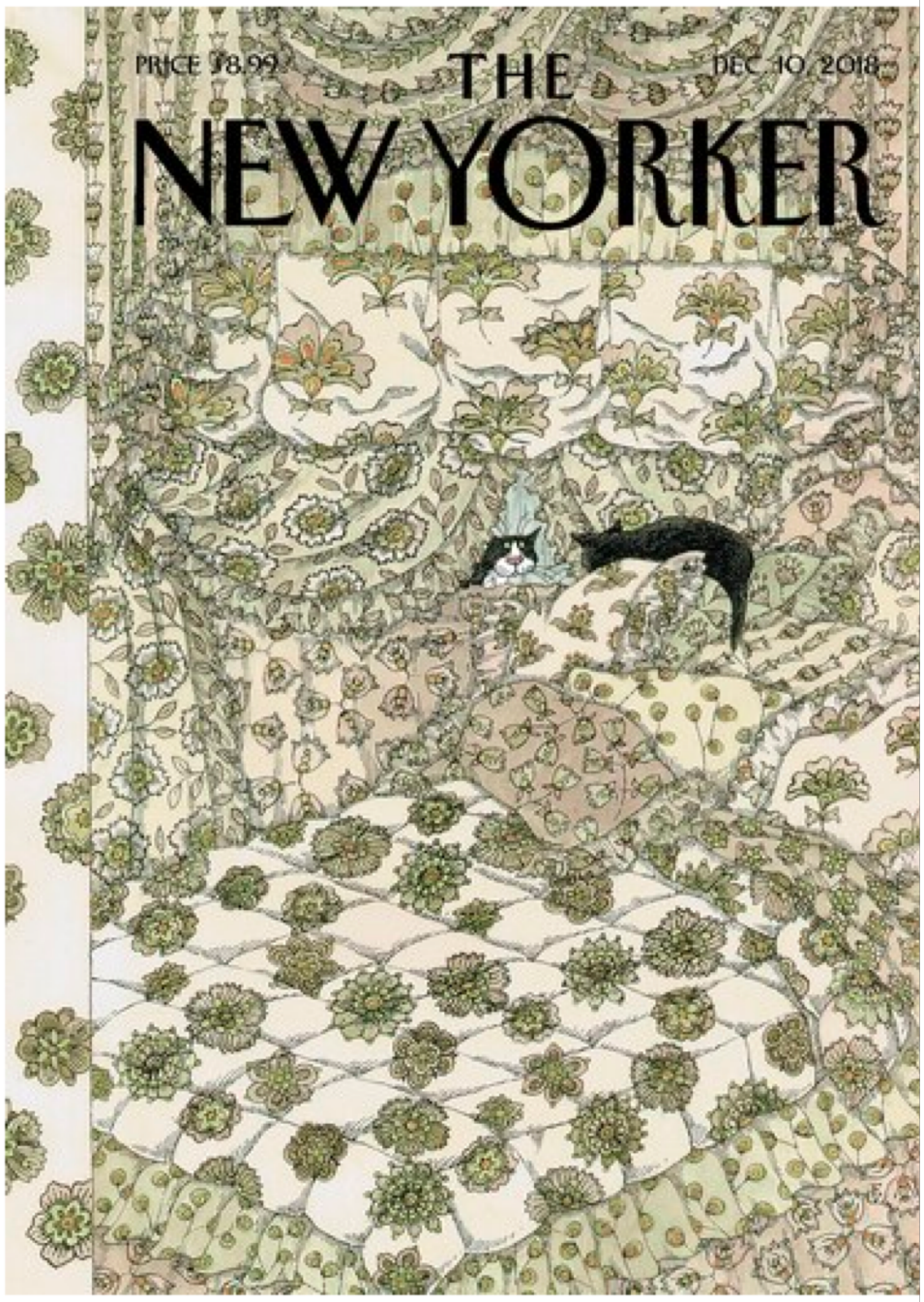
Demo

TWO MONUMENTAL HIGHER-ORDER FUNCTIONS

`map`

`fold`

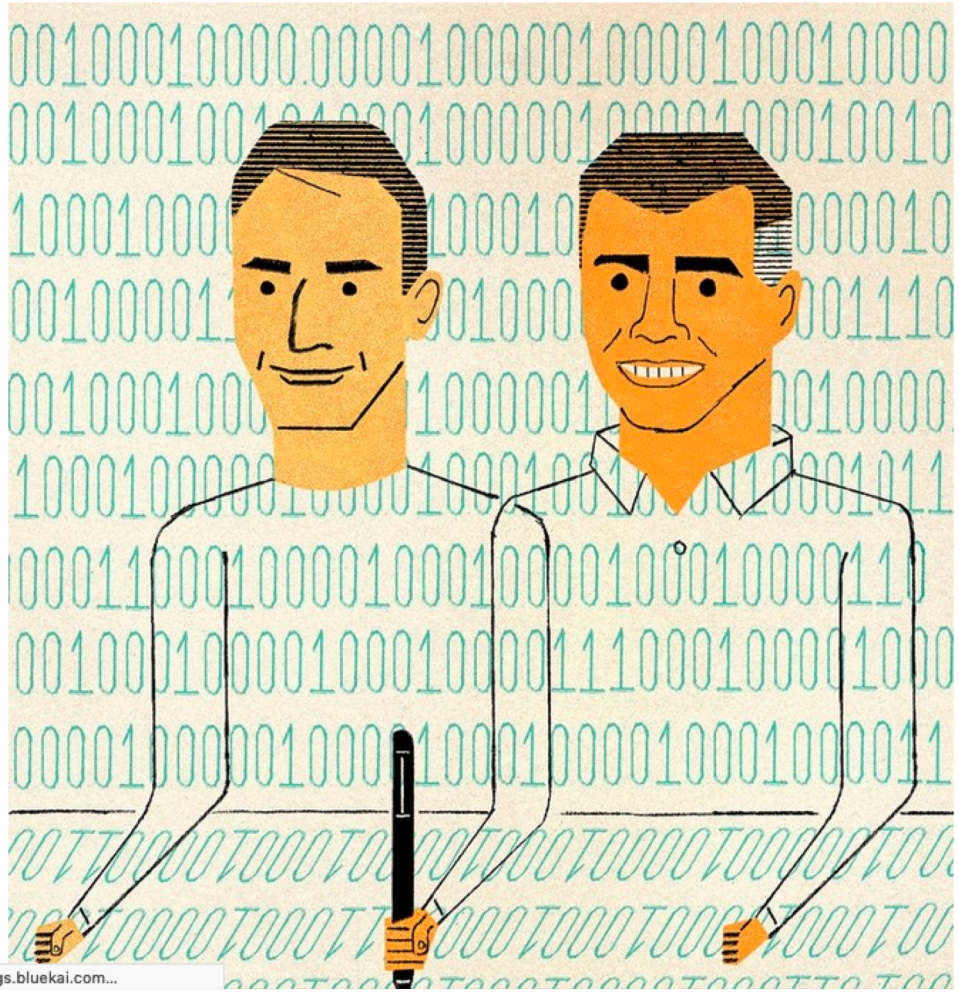
Sibling: **`reduce`**



THE FRIENDSHIP THAT MADE GOOGLE HUGE

Coding together at the same computer, Jeff Dean and Sanjay Ghemawat changed the course of the company—and the Internet.

By James Somers



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]

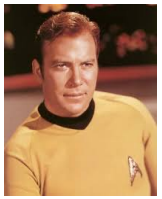
transform list elements

`map`

`fold`

Map

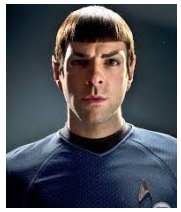
```
map (fun x -> shirt_color(x)) [
```



```
]
```

Map

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

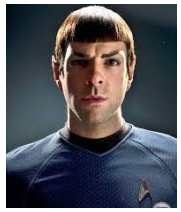

```
]
```

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

Map

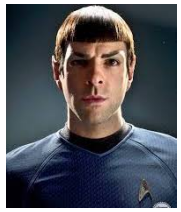
bad style!

```
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
   | [] -> []
   | 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

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"

`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”

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

- [Today] Foster OH Gates 432 1:15-2:15pm
- [Today] Level Up! Gates 310 7-8pm

This is monumental.

THIS IS 3110