Programming Abstractions

Lecture 1: Introduction

Stephen Checkoway

About the course

This is a course about Programming Languages

We will use the language Scheme to discuss, analyze and implement various aspects of programming languages.

Course website: https://checkoway.net/teaching/cs275/2022-spring/

- Contains the syllabus, readings, homeworks, and slides

Office hours in King 231

- Tuesday 13:30–14:30
- Friday 13:30–14:30

Parts of the course

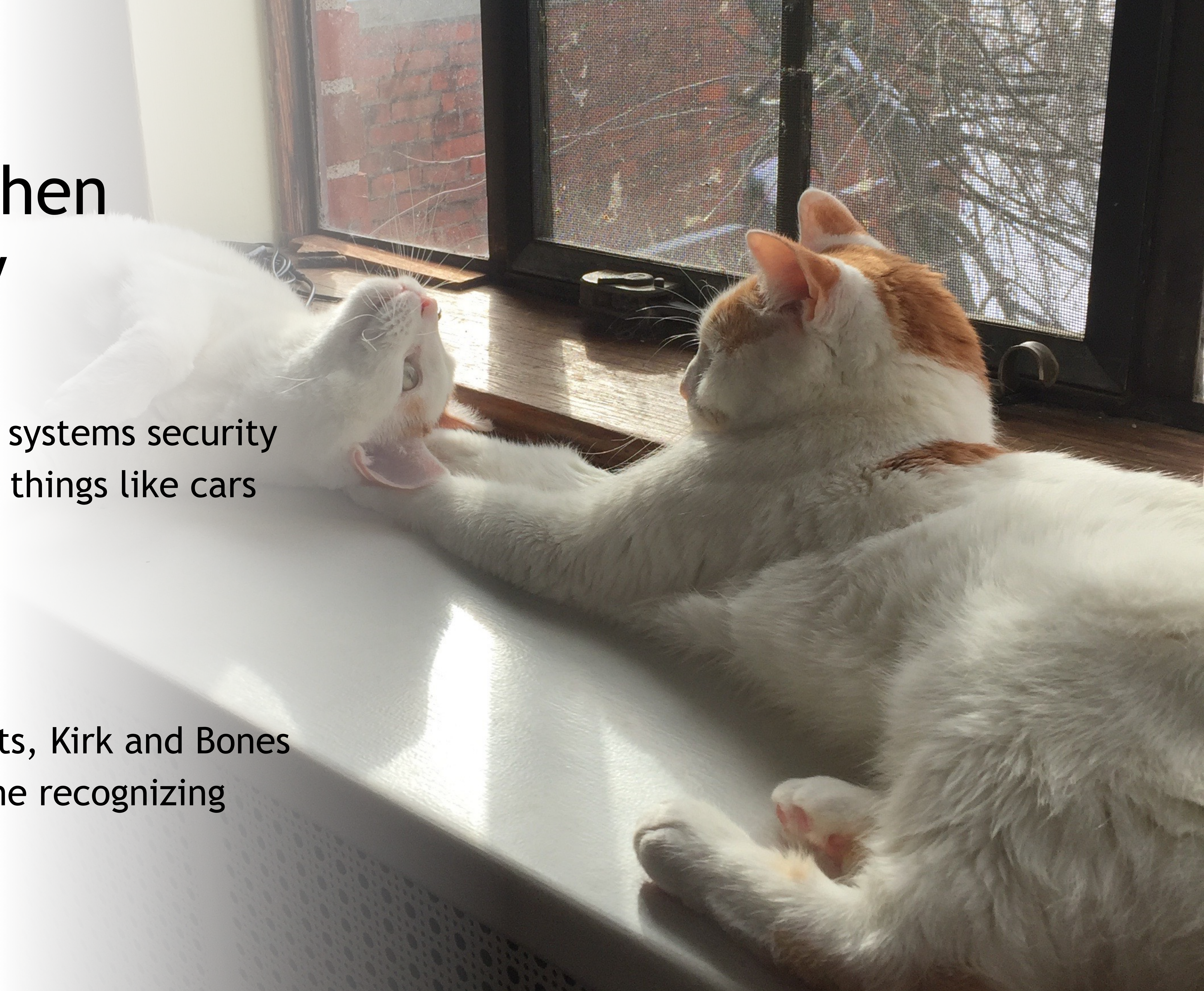
Scheme and things you can do with it (5 weeks)

Implementing Scheme and other languages (4 weeks)

Advanced issues (delayed evaluation, continuations, etc.) (2 weeks)

~~Logic Programming, and Prolog or Racklog (maybe) (2 weeks)~~

Who am I?
Professor Stephen Checkoway



- Research:
  - Computer/Embedded systems security
  - Hacking computers in things like cars and planes

- Fun Facts:
  - I enjoy picking locks
  - I have two Oberlin cats, Kirk and Bones
  - I have a *very* hard time recognizing faces

A quick history of Scheme

John McCarthy invented LISP at MIT around 1960 as a language for AI.

LISP grew quickly in both popularity and power. As the language grew more powerful it required more and more of a system's resources. By 1980 5 simultaneous LISP users would bring a moderately powerful PDP-11 to its knees.

Guy Steele developed Scheme at MIT 1975-1980 as a minimalist alternative to LISP.

Scheme is an elegant, efficient subset of LISP. It has some nice properties that we will look at that allow it to be implemented efficiently. For example, most recursions in Scheme turn into loops.

Why Scheme for CS 275?

All LISP-type languages have lists as the main data structure

- Programs are lists
- Data are lists
- Scheme programs can reason about other programs. This makes Scheme useful for thinking about programming languages in general.

Scheme is a different programming paradigm

- Python, Java, C and other languages are imperative languages. Programs in these languages do their work by changing data stored in variables
- Scheme programs can be written as functional programs—they compute by evaluating functions and avoid variable assignments.

Why Scheme for CS 275?

Scheme is very elegant. It is much less verbose than Java, which means it is easier to see what is happening in a Scheme program.

It is fun!

Assessment

Eight homeworks

- Between about 7 and 10 days per homework
- You can work by yourself or in groups of 2
- Each has lots of small, independent parts
- Three free late days to use throughout the semester

Two midterms exams

One final exam (optional)

Class participation

Clickers!

- Lets you vote on multiple choice questions in real time.
- You need one by next Friday

Clicker poll questions

Peer instruction

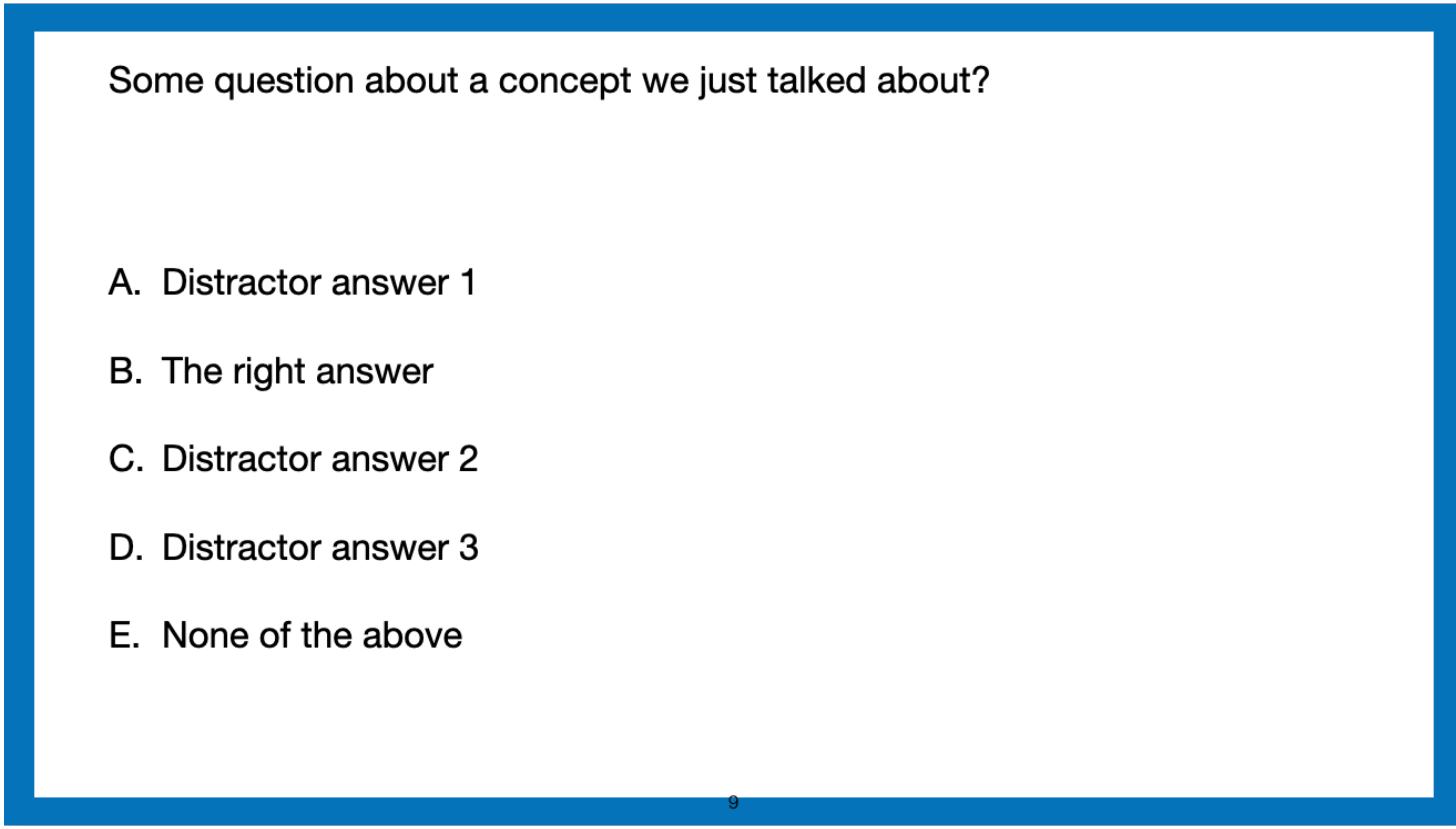
I'll read the question

Answer the poll individually

Group discussion, come to consensus

Everybody in group votes the consensus

Report your group's vote/thinking



Some question about a concept we just talked about?

A. Distractor answer 1

B. The right answer

C. Distractor answer 2

D. Distractor answer 3

E. None of the above

9

Group Discussion Norms

Make sure everyone gets to talk

Have everyone state their answer before discussing which answer is correct

Take turns reporting out

If you think someone is wrong, ask them to explain their thinking rather than just dismissing it

Class Norms

Contribute as you feel comfortable

- If you're not comfortable answering, you can pass.
- If you're not usually inclined to speak much in class, push yourself to ask questions more often.

Be aware of the space you take up in class

- Make space for others, use some space for yourself

The main goal of every person in the class should be to engage proactively with the ideas we understand the least. If someone asks a question/makes a comment that seems obvious to you, show them respect.

Scheme interpreter: DrRacket

Racket is Scheme plus extra nice stuff

- One consequence is Scheme has a bunch of traditional names for list functions that are bad, Racket has better names! We'll learn and use both as appropriate

We're actually going to be using Racket in this course

- I'm probably going to use Racket and Scheme interchangeably (sorry)

DrRacket is free https://www.racket-lang.org

Todo this week

Readings from *Racket Programming The Fun Way*

- Chapter 1 through "Symbols, Identifiers, and Keywords"

Readings from *The Racket Guide*

- Section 4.10

Install DrRacket before next class

Start Homework 1

- Due Friday, March 4 at 23:59

Introducing Scheme

Expression in Scheme (s-expression)
(Traditional)

A symbolic expression (s-expression) is one of the following

- An atom
  - A number, e.g., `5`, `-10`, `8.3`
  - Boolean values `#t` and `#f`
  - A string, e.g., `"foo"`
  - A symbol, e.g., `'foo`, `'list-ref`, `'pair?`, `'set!`
- Null
  - Written `null` or `'()`
- A pair
  - Written `(x . y)` where `x` and `y` are s-expressions
- A variable, e.g., `foo`, `list-ref`, `pair?`

Expressions in Racket

(Modern)

The concept of an atom isn't as meaningful now

Racket adds additional data types that aren't pairs, aren't null, and aren't really atoms (like vectors)

For the most part, we're going to ignore these extra data types in this course

Arithmetic/logical/string operations

3 + 5	`(+ 3 5)`
$x \bullet (4 + y + z)$	`(* x (+ 4 y z))`
x AND y	`(and x y)`
x OR y OR z	`(or x y z)`
"hello" + " " + "world"	`(string-append "hello" " " "world")`

In C, Python, or Java, we would compute the arithmetic mean (average) of two numbers (or variables holding numbers) as `(x + y) / 2`. How do we do this in Scheme or Racket?

A. `(x + y) / 2`

B. `((x + y) / 2)`

C. `(+ x y / 2)`

D. `(+ (/ x y) 2)`

E. `(/ (+ x y) 2)`

Lists

Lists are the most important data type in Scheme

A list is one of two things

- `null`, the empty list
- A pair `(x . y)` where `x` is an s-expression and `y` is a list
  - `x` is called the head of the list and `y` is the tail

This is a recursive type definition: a type defined in terms of itself!

Special syntax for lists

`'(1 2 3 4)` is a list of 4 integers

`'(a b c)` is a list of three symbols

`'(#t #f #t #t #f)` is a list of 5 booleans

`'(42 -8 #t + "foo")` is a list of 5 atoms ('+ is a symbol)

It's equivalent to
`'(42 . (-8 . (#t . (+ . ("foo" . ())))))`

Lists are heterogeneous (they can contain elements of different types)

The empty list

There are three ways to write the empty list, they're equivalent

- `null`
- `empty`
- `'()` — We'll see shortly why this has a leading ' like a symbol does

All of these are simply a null pointer

We can use them mostly interchangeably, but when working with lists (as opposed to some other data type we might build out of pairs), using `empty` or `'()` can make it clear you mean the empty list specifically

Creating a list

`(list)` produces the empty list `'()`

- `null`, `empty`, and `'()` also do this

`(list 1 3 5 2)` produces the list `'(1 3 5 2)`

`(list #t 5 "foo")` produces the list `'(#t 5 "foo")`

`(list (* 2 3) (and #t #f) 8)` produces `'(6 #f 8)`

Quoting

Placing a `'` before an s-expression "quotes" it

- The quoted expression is treated as data, not code
- DrRacket displays lists with the quote

`'(1 4 5)` is a 3-element list

We saw `(list (* 2 3) (and #t #f) 8)` produces `'(6 #f 8)`

`'((* 2 3) (and #t #f) 8)` produces `'((* 2 3) (and #t #f) 8)`

- This is a 3 element list:

```
'((* 2 3)                 ; 1st element, itself a 3-element list
  (and #t #f)             ; 2nd element, another 3-element list
  8)                      ; 3rd element, the number 8
```

Quoting

Quoting a number, boolean, or string returns that number, boolean, or string

- `'35` gives `35`
- `'#t` gives `#t`
- `'"Hello!"` gives `"Hello!"`

Quoting a variable gives a symbol

- `+` and `string-append` are variables whose values are procedures
- `'+` and `'string-append` are symbols

Quoting a list gives a list of quoted elements

- `'(1 2 x y)` is the same as `(list '1 '2 'x 'y)`
- `'(() (1) (1 2 3))` is the same as `(list '() '(1) '(1 2 3))`

Guidelines for creating lists

If you want to evaluate some expressions and have the resulting values be in the list, use `(list expr1 expr2 ... exprn)`

- E.g., `(list x (list x y z) z)`

If you want to create a list of literal numbers/strings/booleans/symbols, use '(...)

- E.g., `'(10 15 20 -3)`

Given variables x and y, how do we create a list containing the values of x, y, and x+y? I.e., if x is 10 and y is 15, the list we create is `'(10 15 25)`.

A. `(list x y (+ x y))`

B. `(list 'x 'y (+ 'x 'y))`

C. `(list 'x 'y '(+ x y))`

D. `'(x y (+ x y))`

E. All of the above

Some examples

Given the code

```
#lang racket

(define x 100)
(define y 42)
(define z 68)

(define hi "Hello")
(define bye "Goodbye")
```

Let's compute

- x + y + z
- (x + y + z) / 3
- appending the values of hi and bye
- a list containing the symbol 'x and the value of x