Racket:
Lists & Recursion
Your participation

Please describe your level of agreement with this statement:

I participate in class a lot.

1 = strongly agree; 4 = neutral; 7 = strongly disagree

and write a brief note to me that explains your response.
The pace of this class is...
I’m learning a lot in CS 42.
CS 42 is interesting.
I can get help / support

- Strongly disagree
- Neither agree nor disagree
- Strongly agree
When it comes to workload, so far, this is my hardest course
The sketches are useful
What can I (the student) do?

- start assignment early
- office / tutoring hours
What’s working / not working?
and what can I (the instructor) do to help?

- assignments
- collaboration
- course content
- sketches
- week-long assignments
- broccoli ice cream
- week-long assignments
- broccoli ice cream
- more depth
- time management
What’s working / not working?
and what can I (the instructor) do to help?

more platypuses!
Let’s practice: Racket functions

(see handout)
The Racket debugger is here to help
Creating and accessing Racket lists

\[(\text{cons} \ 1 \ '(2 \ 3))\]
Watch out!

Don’t do these things (and if you accidentally do, know how to recognize them)

> (cons 1 2)
'(1 . 2) ← not a list!

This expression builds a pair.

A pair is not a list.

You can’t call first on it.
You can’t call rest on it.

> '(list 1 2)
'(list 1 2)

This expression builds a list whose first element is 'list!

For more info, see: docs.racket-lang.org/guide/Pairs_Lists_and_Racket_Syntax.html
Inductive Data Types & Recursive Operations
(Putting Together & Taking Apart)

empty
cons
empty?
first
rest
Recursive functions over lists

Note: Racket has a built-in length function. For this example only, we’re pretending it doesn’t exist.

;;; len
;;; inputs: a list, L
;;; outputs: the number of elements in the list
(define (len L) )
Recursive functions over lists

Note: Racket has a built-in length function. For this example only, we’re pretending it doesn’t exist.

;;; len
;;;   inputs: a list, L
;;;   outputs: the number of elements in the list
(define (len L)
  (if (empty? L) 0 (+ 1 (len (rest L)))))

;; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
Recursive functions over lists

Note: Racket has a built-in length function. For this example only, we’re pretending it doesn’t exist.

```scheme
;; len
;;  inputs: a list, L
;;  outputs: the number of elements in the list
(define (len L)
  (if (empty? L) 0 (+ 1 (len (rest L)))))

; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
```

base case

recursive step
Recursive functions over lists

Note: Racket has a built-in length function. For this example only, we’re pretending it doesn’t exist.

```scheme
;; len
;;   inputs: a list, L
;;   outputs: the number of elements in the list
(define (len L)
  (if (empty? L)
      0
      (+ 1 (len (rest L))))
)

; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '(((1 2 3)))) 1)
```
Recursive functions over lists

Note: Racket has a built-in length function. For this example only, we’re pretending it doesn’t exist.

;;; len
;;; inputs: a list, L
;;; outputs: the number of elements in the list
(define (len L)
  (if (empty? L)
      0
      (+ 1 (len (rest L))))
  )

; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '(((1 2 3)))) 1)
Recursive functions over lists

Note: Racket has a built-in length function. For this example only, we’re pretending it doesn’t exist.

;;; len
;;; inputs: a list, L
;;; outputs: the number of elements in the list
(define (len L)
  (if (empty? L)
      0
      (+ 1 (len (rest L))))

; tests
(check-equal? (len '()) 0)
(check-equal? (len '(1 2 3)) 3)
(check-equal? (len '((1 2 3))) 1)
Let’s practice: lists + recursion

empty
cons
empty?
list?
first
rest

(see handout)