Modeling with Mutable Data

Session 18

Russell Mirabelli 4/23/2020

"What's with the new layout?"

— you, maybe

Keeping Engaged

Thanks for engagement
I do appreciate it

- While I can’t see how each of you is interacting with the course, I appreciate how you’re taking things seriously.
- It can be difficult to retain focus when studying remotely
- Our typical coursework closes today
- But class continues!

Live Coding

- Next week, we'll be doing "live coding" during class
- I'll be working through and explaining extensions of our interpreter
- I'll encourage you to pause the lecture and stay caught up on entering & testing your code at the same time

Final Exam

You were going to ask...

- The final exam will cover all material since exam 2; it will not be comprehensive
- The final exam will be open-book and administered through BrightSpace
- I have not yet heard university policy on the timing of final exams

Mutable Data

Additions to compound data

- Previously dealing with compound data we used *selectors* and *constructors*
- Considering data to be *mutable* requires a new type of procedure, the *mutator*
- Data objects for which mutators are defined can be called *mutable data objects*
- We're going to use *pairs* and *lists* to generally explore the concepts behind mutating data

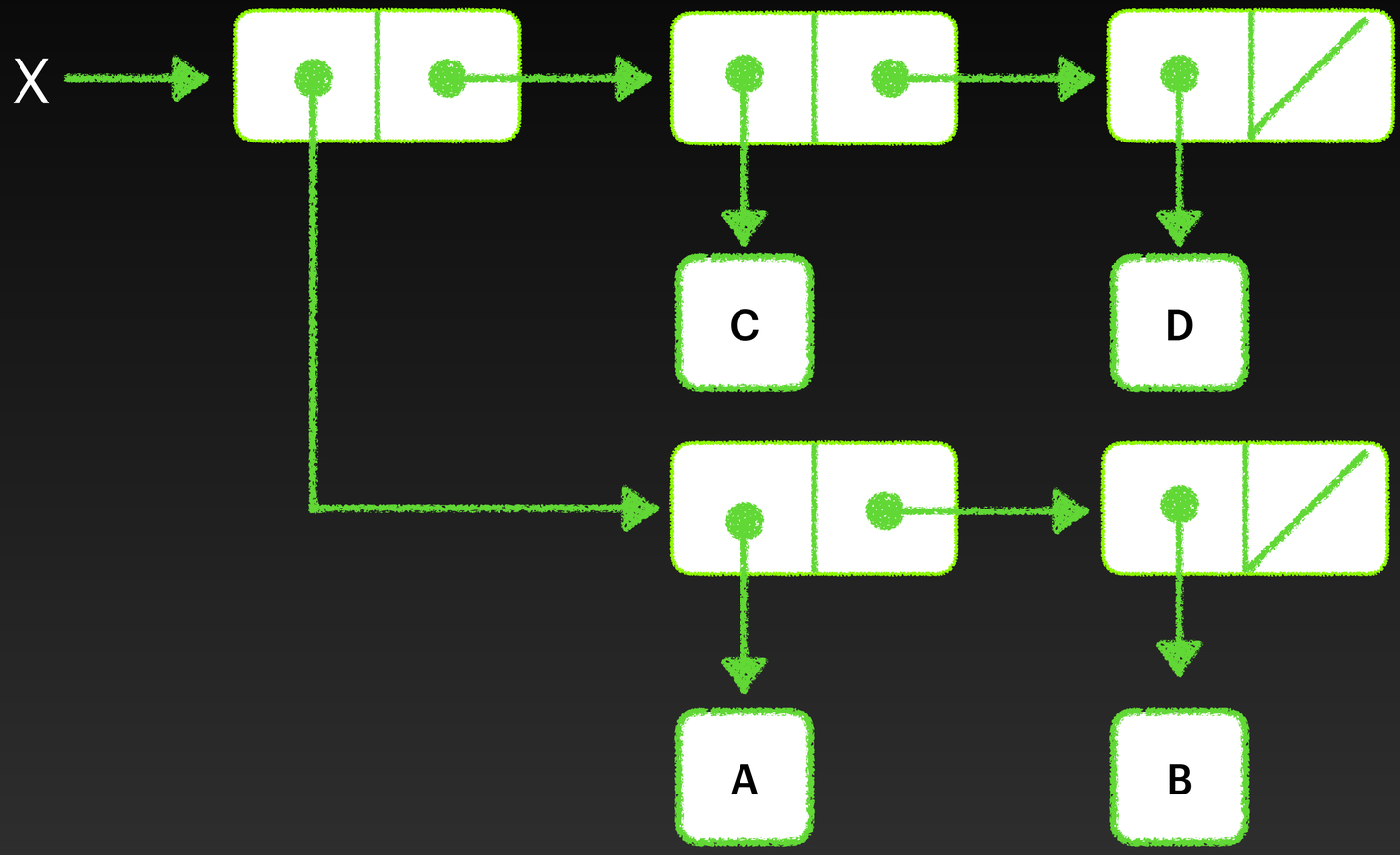
- Pair selectors and constructors:
  - car
  - cdr
  - cons
- List selectors and constructors
  - list
  - append

- The list selectors & constructors can be defined in terms of the pair selectors and constructors
- Modifying a list requires *new* mutators
  - *set-car!*
  - *set-cdr!*
- (The exclamation marks are not for emphasis…)

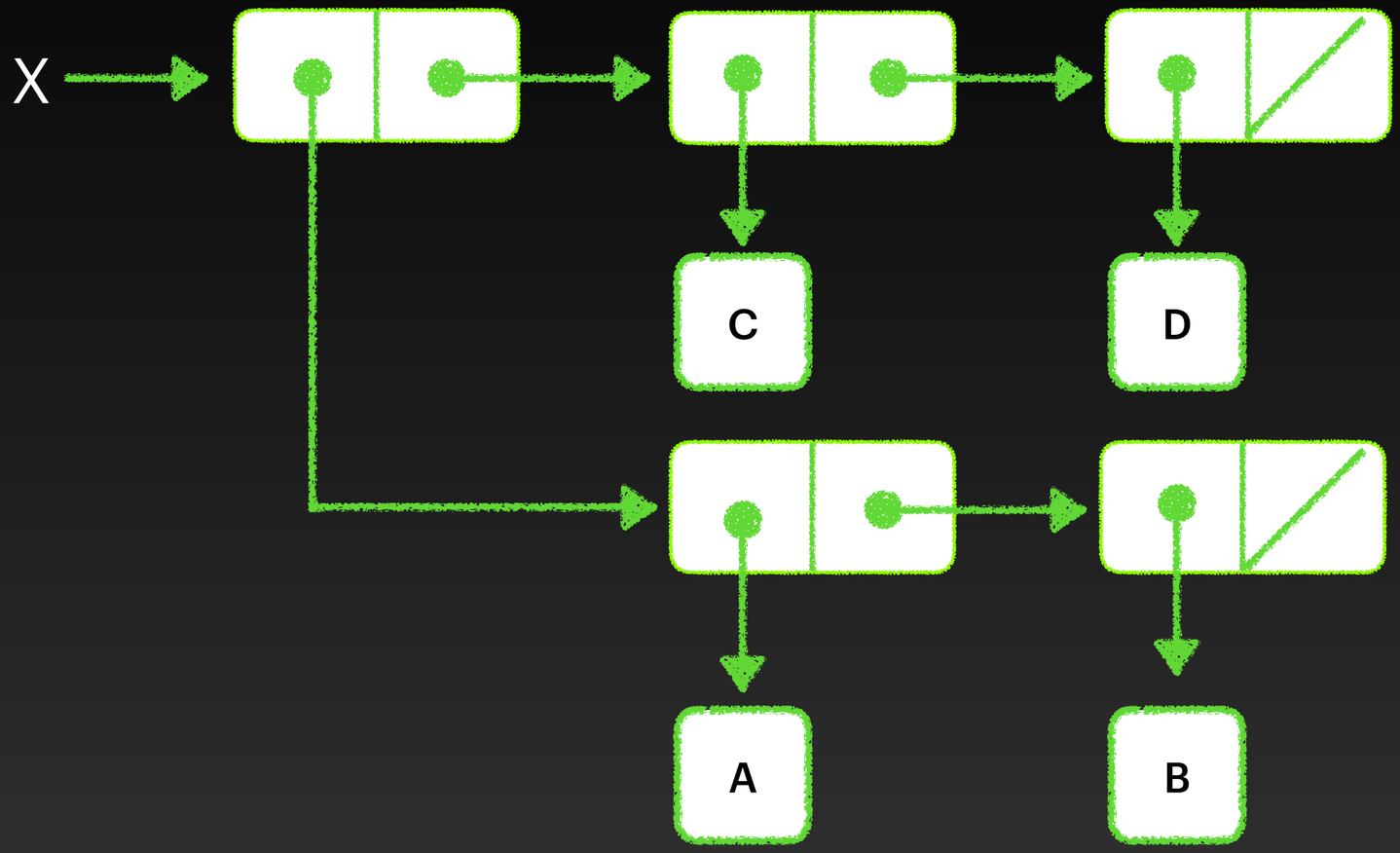
Using set-car! / set-cdr!

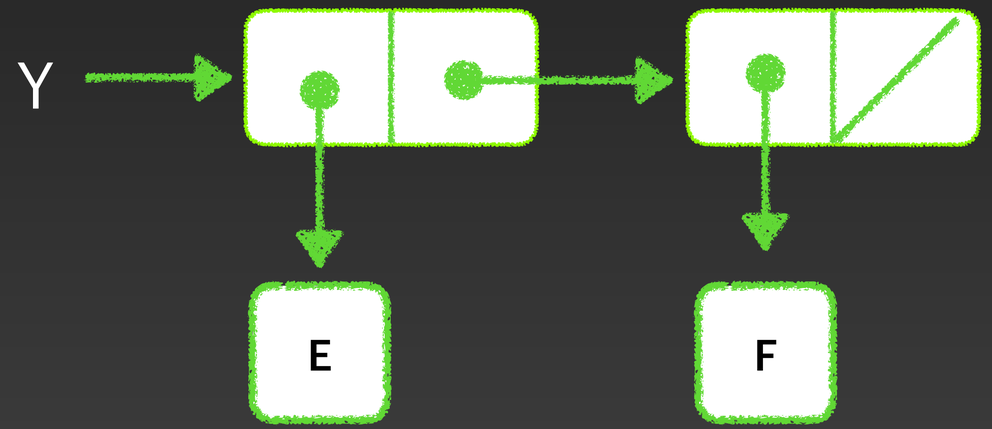
- Both of these mutators take the same form: a pair, and the entity to place in the appropriate part of the original pair
  - *set-car!* replaces the lefthand component (of course)
  - *set-cdr!* replaces the righthand component

set-car! example

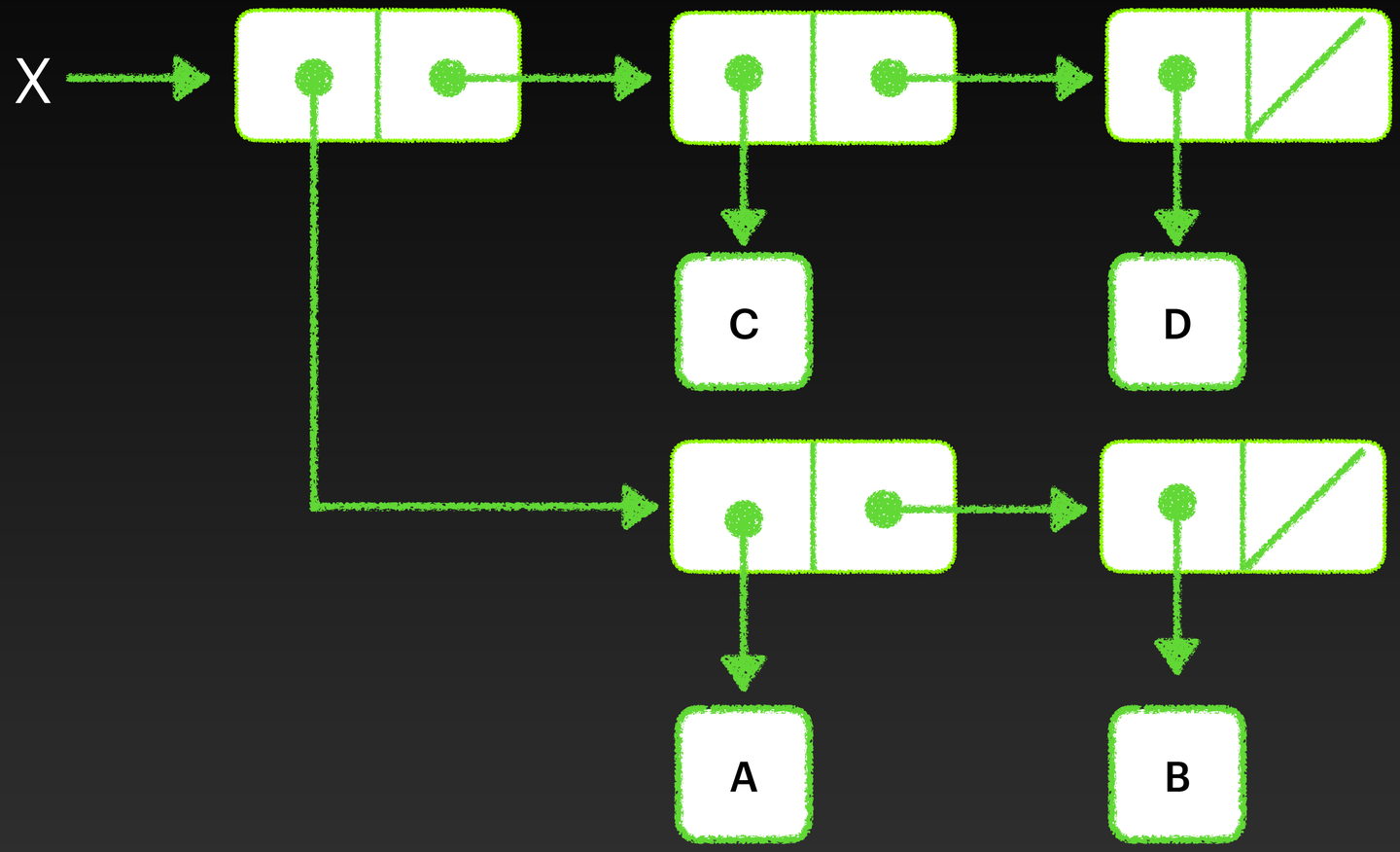


set-car! example



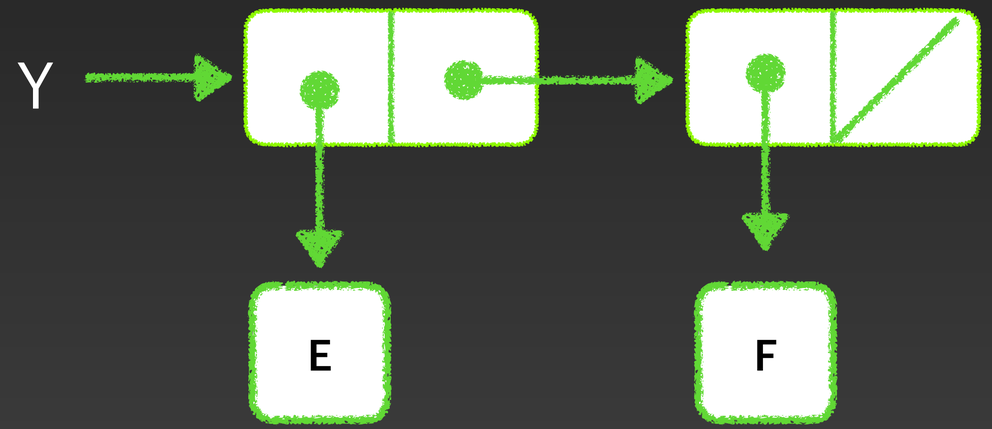


set-car! example



X: ((a b) c d)

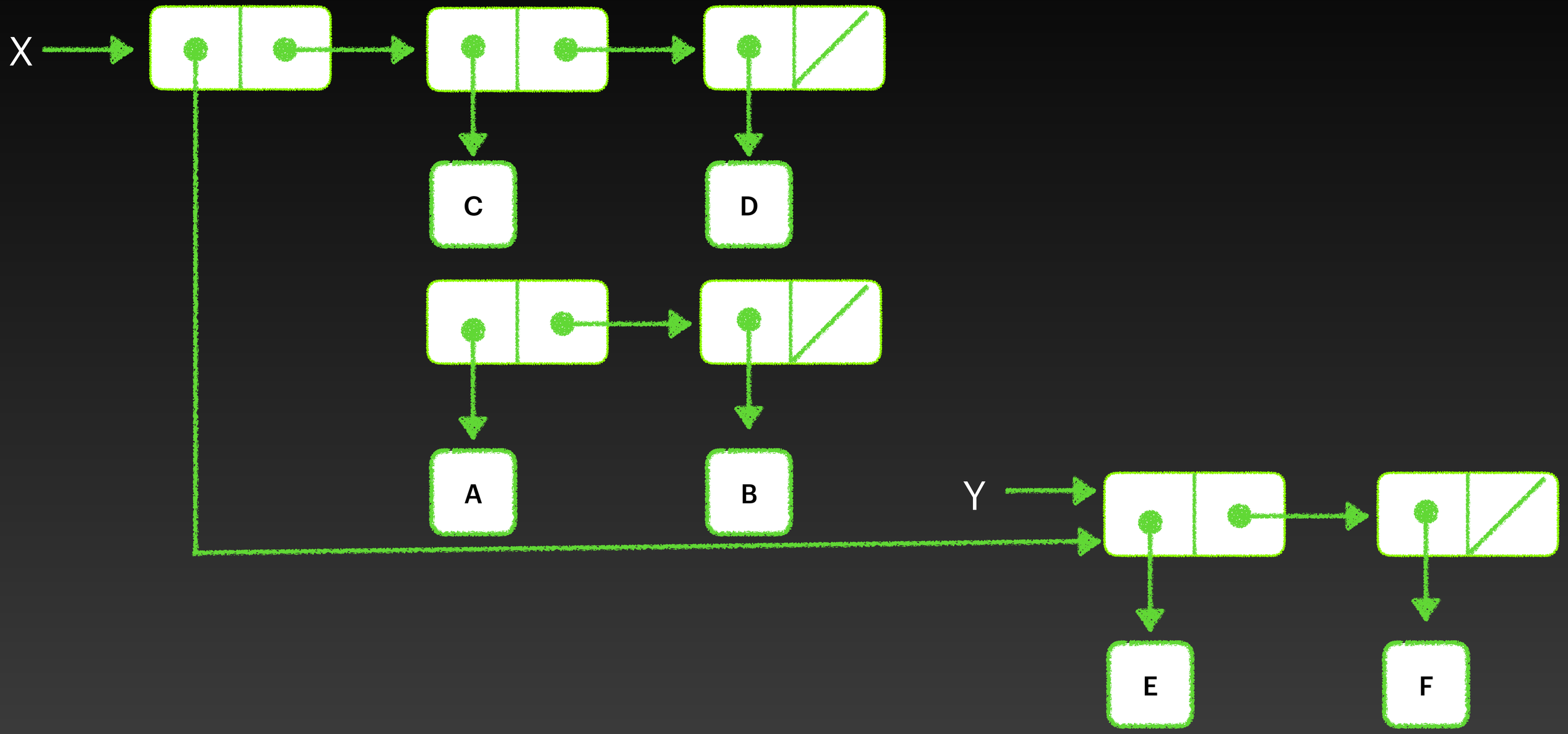
Y: (e f)



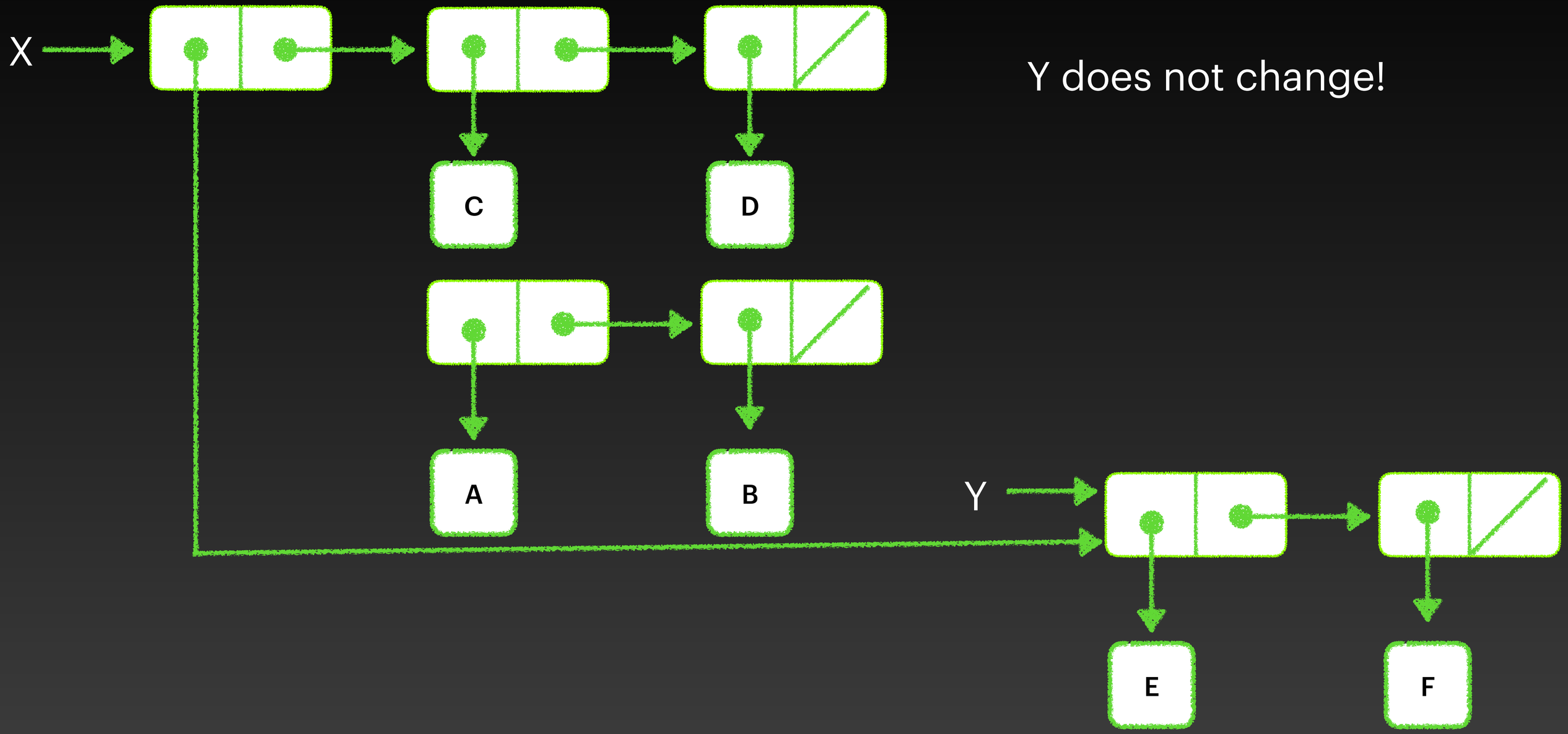
What happens...

If we (set-car! x y)

set-car! example

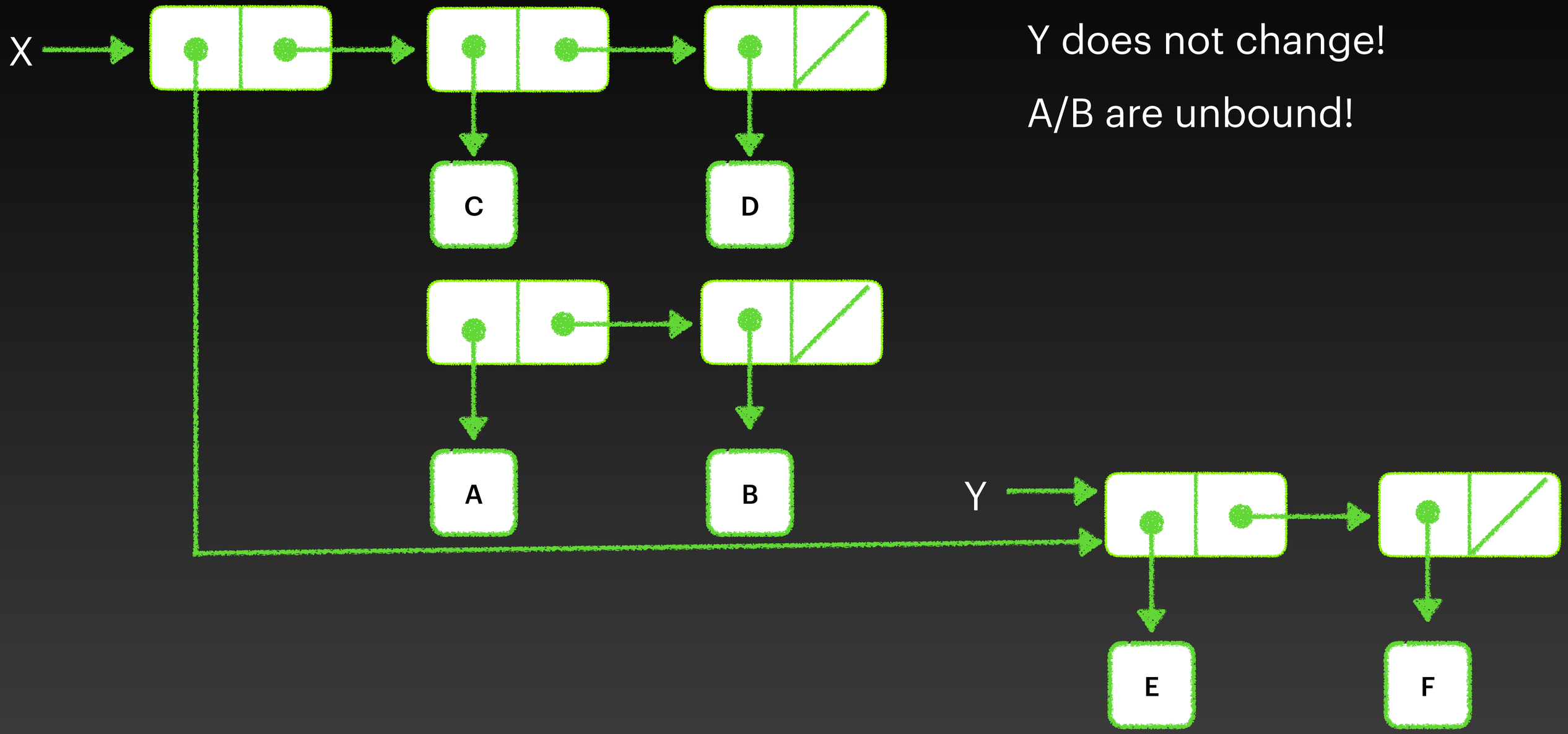


set-car! example



Y does not change!

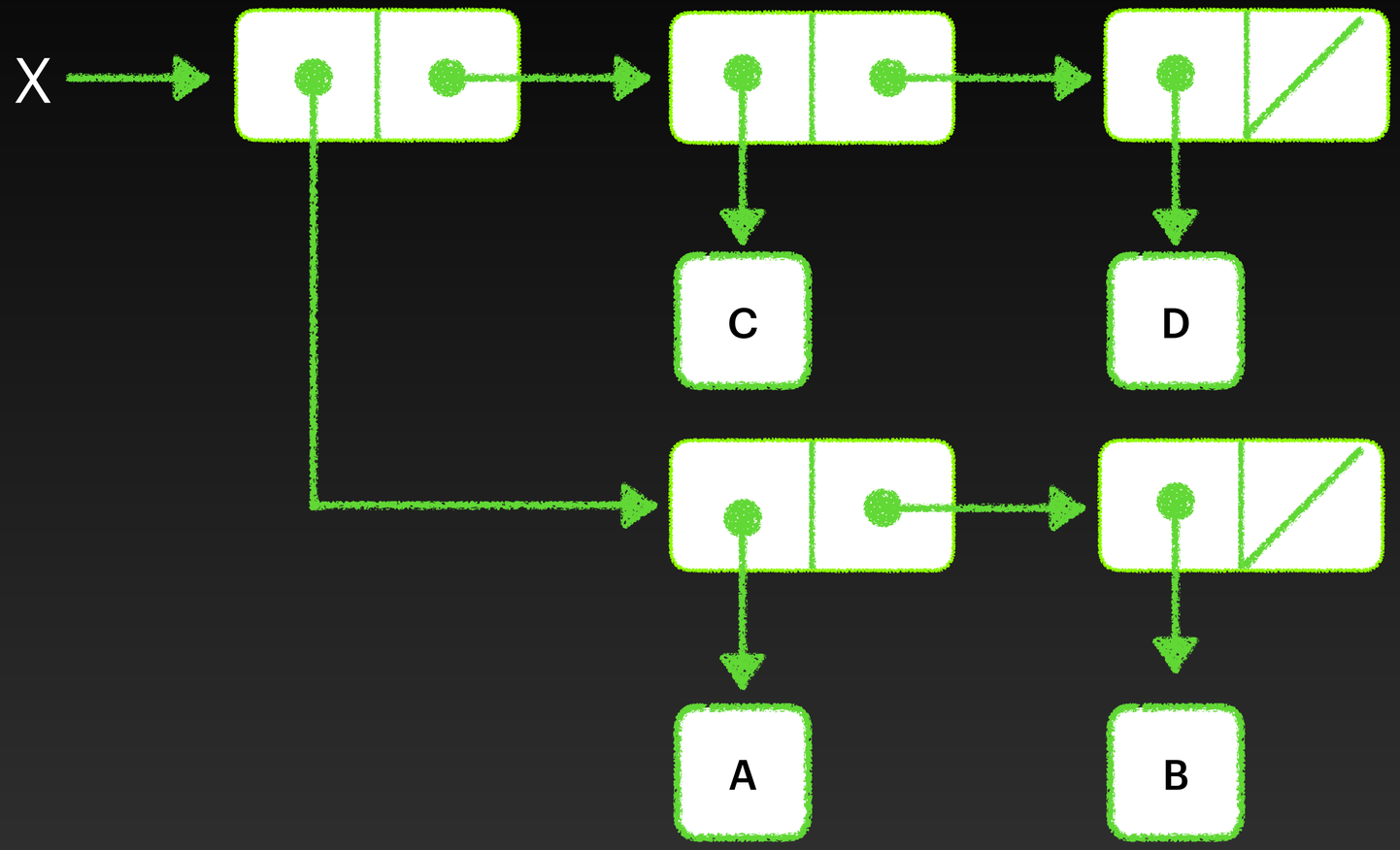
set-car! example



Y does not change!

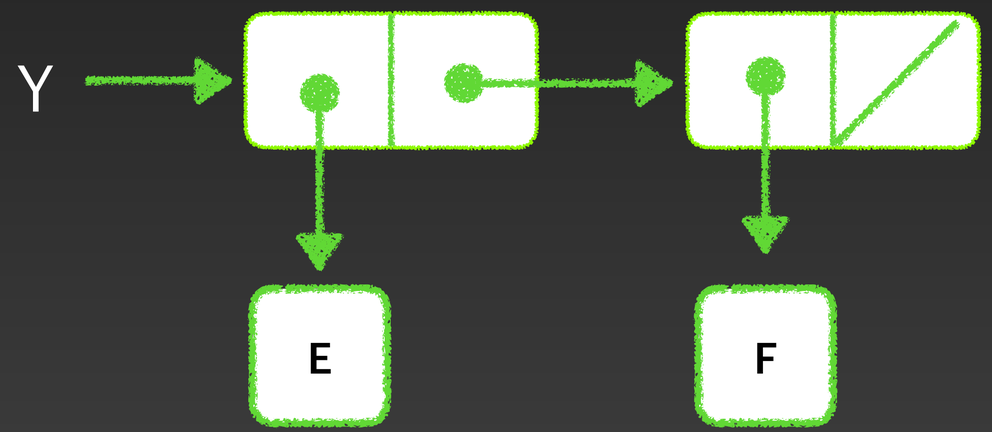
A/B are unbound!

cons example

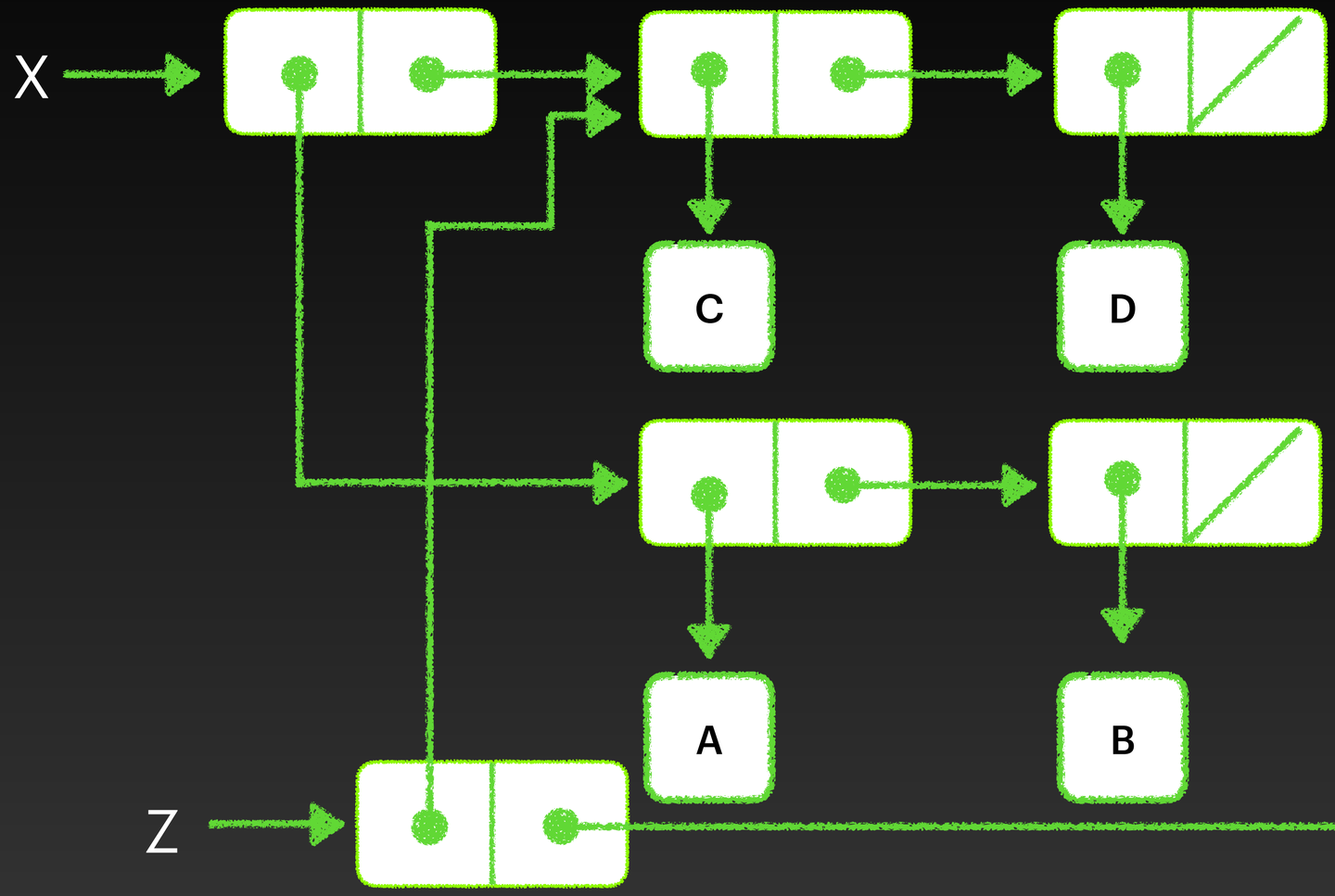


Now let’s try:

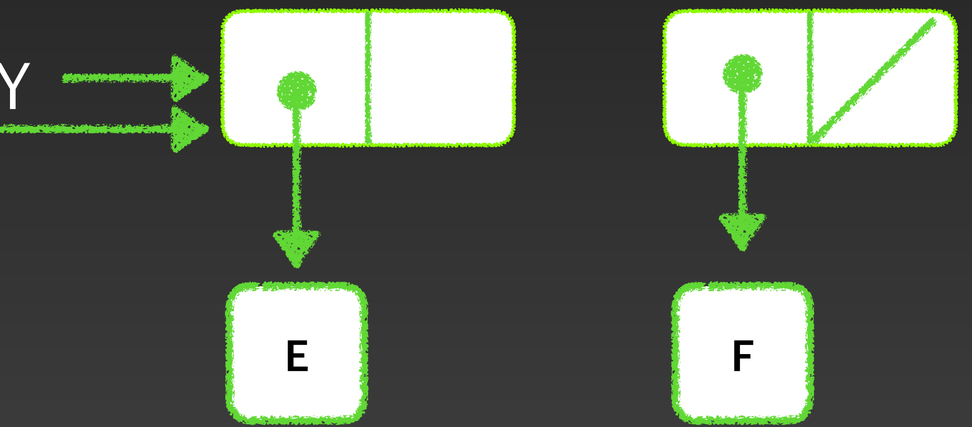
(define z (cons y (cdr x)))



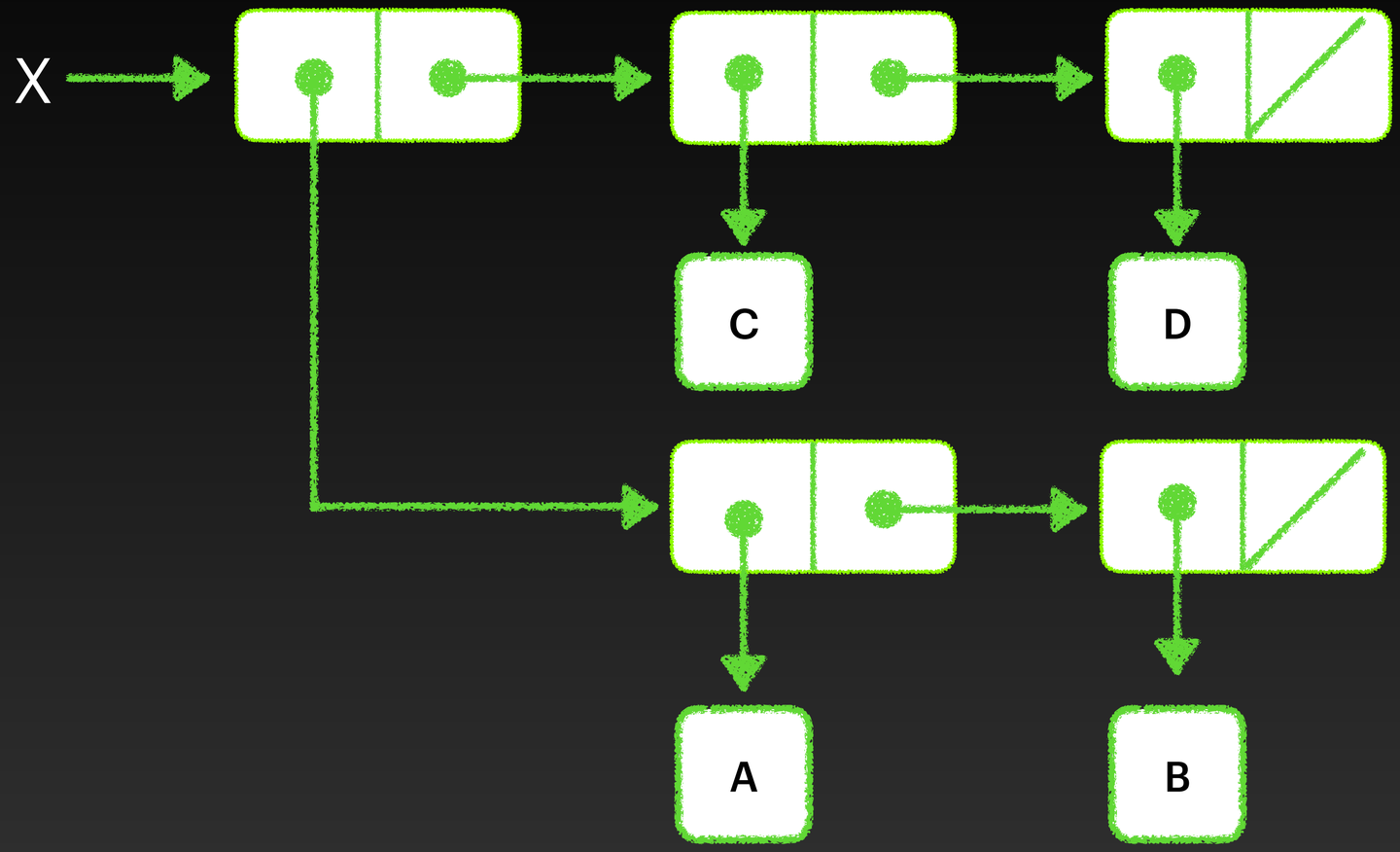
cons example



Note that (a b) never becomes unbound, and x does not change

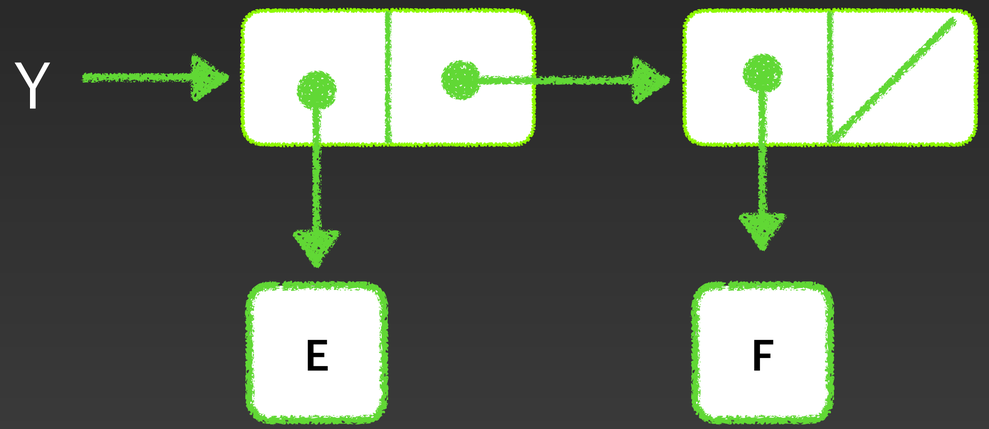


set-cdr! example

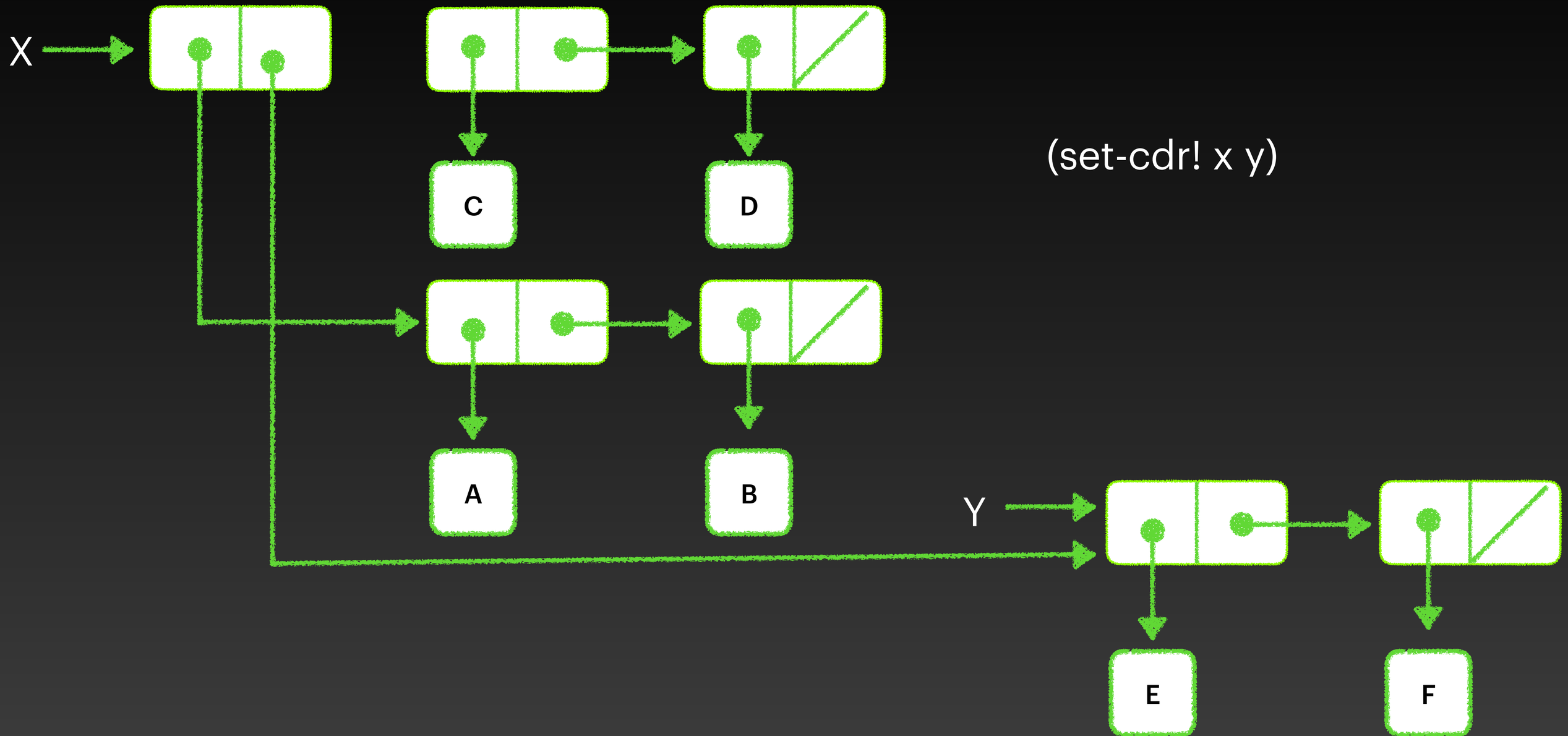


X: ((a b) c d)

Y: (e f)

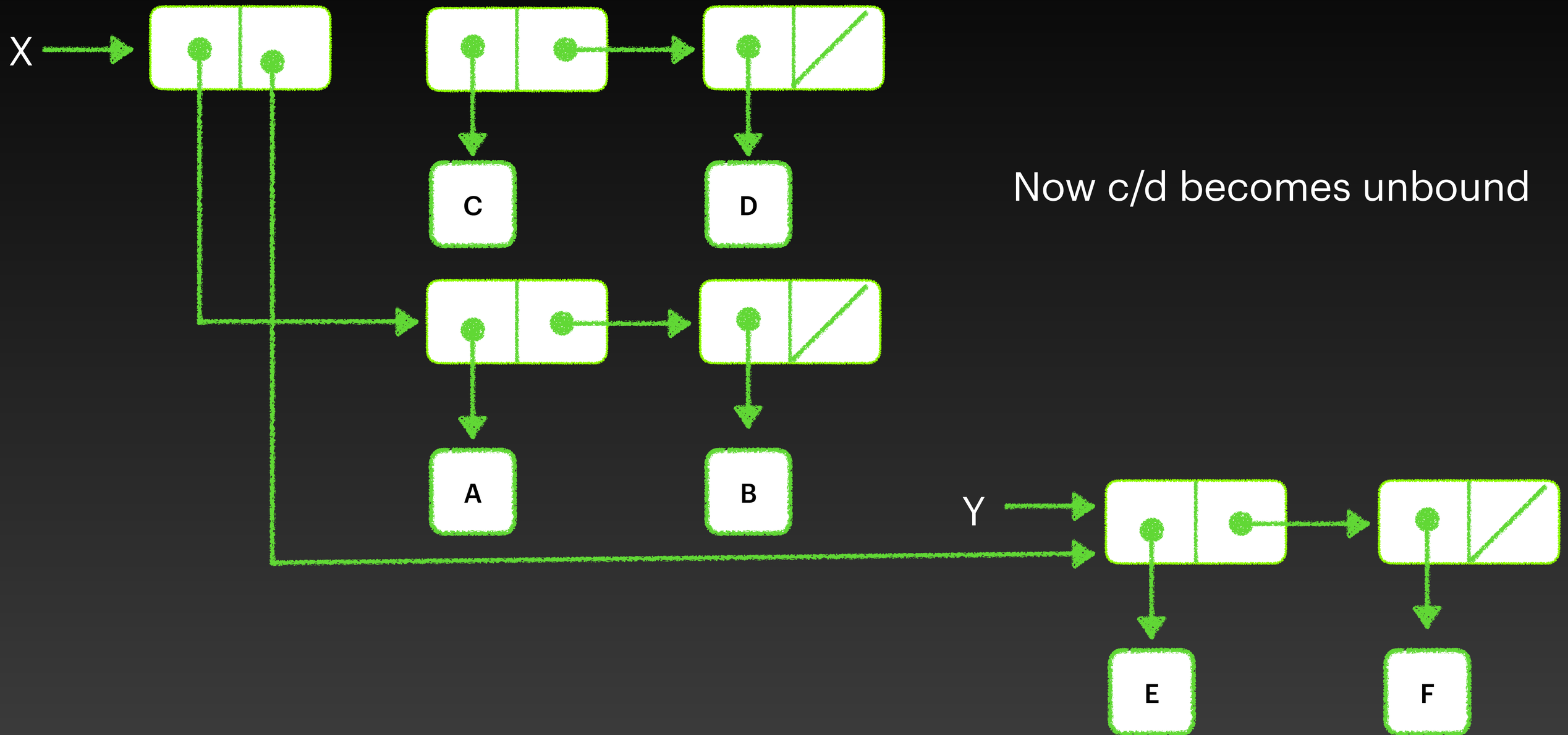


set-cdr! example



(set-cdr! x y)

set-cdr! example

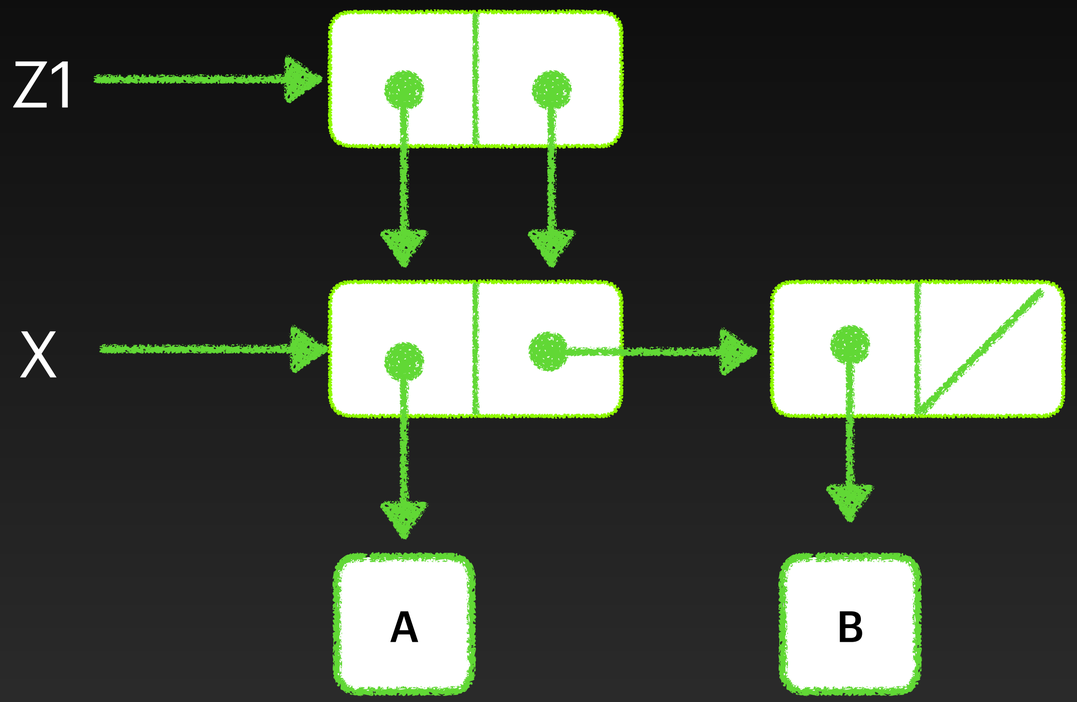


Now c/d becomes unbound

Sameness

- Two elements are considered the same if they’re connected to the same structure
- Generally, if we point to a constructed entity multiple times, then we have two identical references. If we construct those pointers multiple times, then we have two non-identical references.

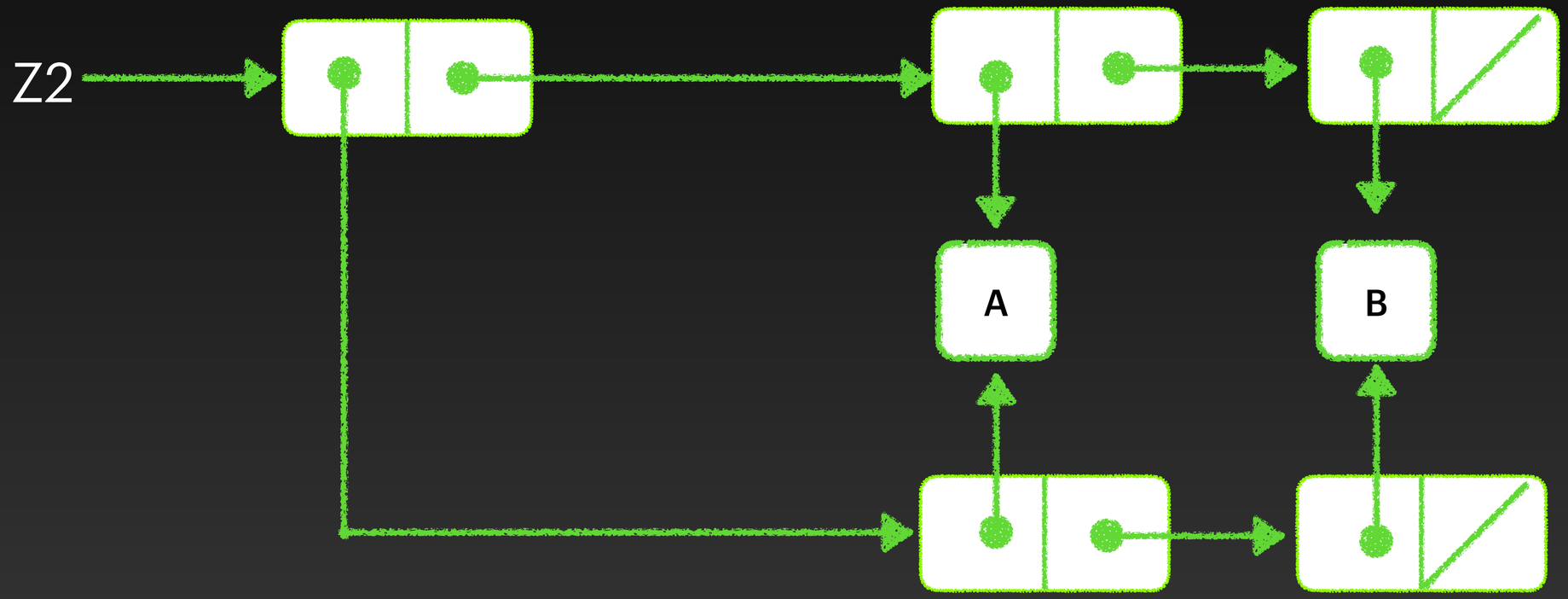
Identical References




```
(define x (list 'a 'b))
(define z1 (cons x x))
```

Distinct References

(define z2 (cons (list 'a 'b) (list 'a 'b)))



The pairs of the two lists are distinct, but point to same contents

Sharing

- These issues we see with sharing are the reason that we need to take care with assignment
  - Unintended copies may be made
  - Data may be orphaned

Designing a queue

Designing a queue

Have you ever noticed that the letters in queue are just lining up to make a 'q' sound?

Queues are standard constructs

and you've been using lots of them

A queue is a data structure with two fundamental operations:

- Insert items at the tail of a list
- Remove items from the head of a list

This of course assumes a FIFO queue, but that's a reasonable assumption

(You have probably studied queues in data structures. Please bear with me)

- Constructing a queue needs two low-level operations: *set-car!* and *set-cdr!*
- A queue cannot be built with simply car, cdr, and cons
- Let’s look at a sample queue

(define q (make-queue))	
(insert-queue! q 'a)	a
(insert-queue! q 'b)	a b
(delete-queue! q)	b
(insert-queue! q 'c)	b c
(insert-queue! q 'd)	b c d
(delete-queue! q)	c d

Queue operations

- (make-queue) returns an empty queue

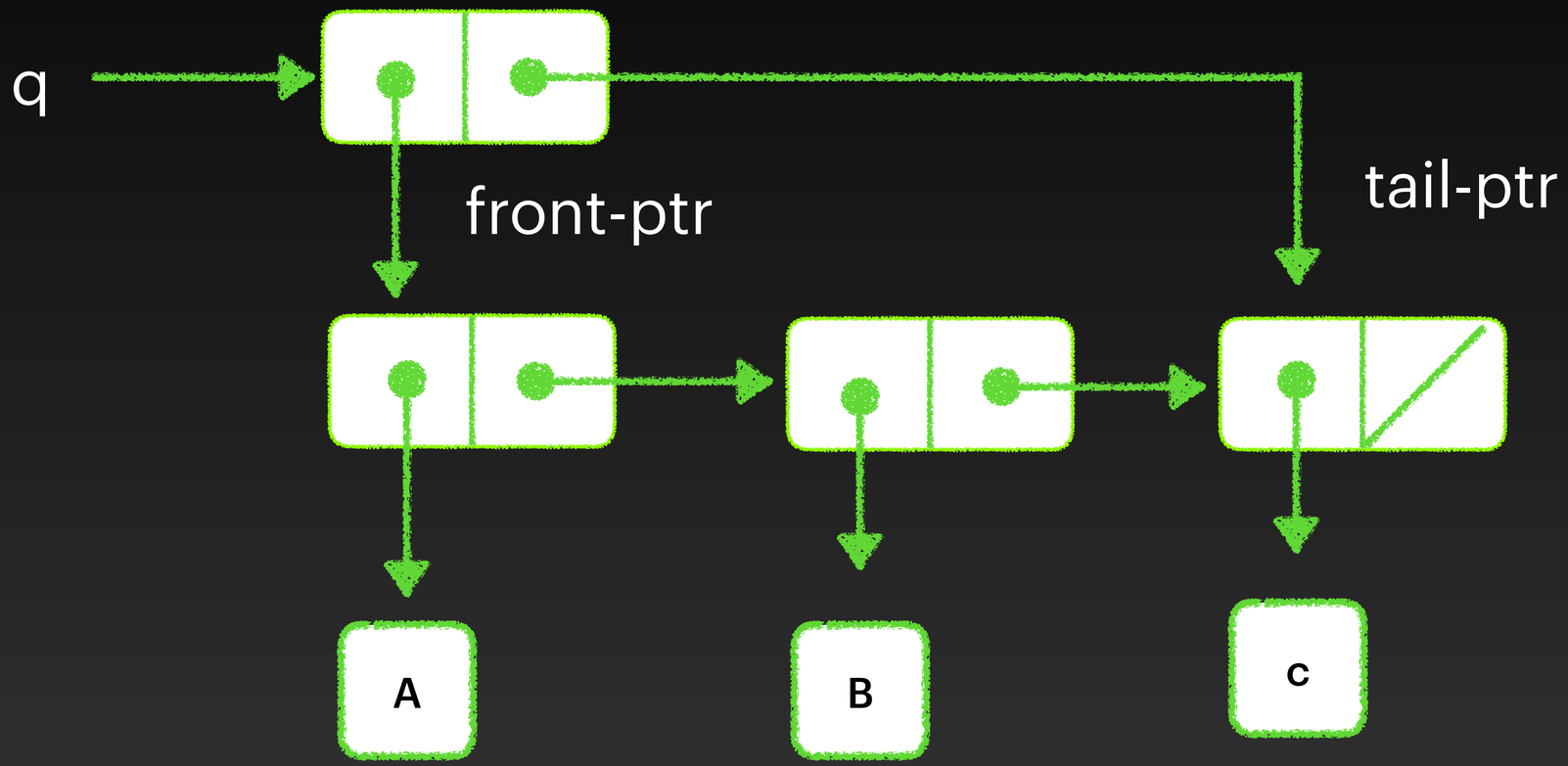
- (empty-queue? *queue*) returns true if the queue is empty
- (front-queue? *queue*) returns the first object in the queue, or an error if the queue is empty. It *does not modify* the queue

- (insert-queue! *queue element*) adds an element to the tail of the queue
- (remove-queue! *queue)* removes the frontmost element from the queue

List representation

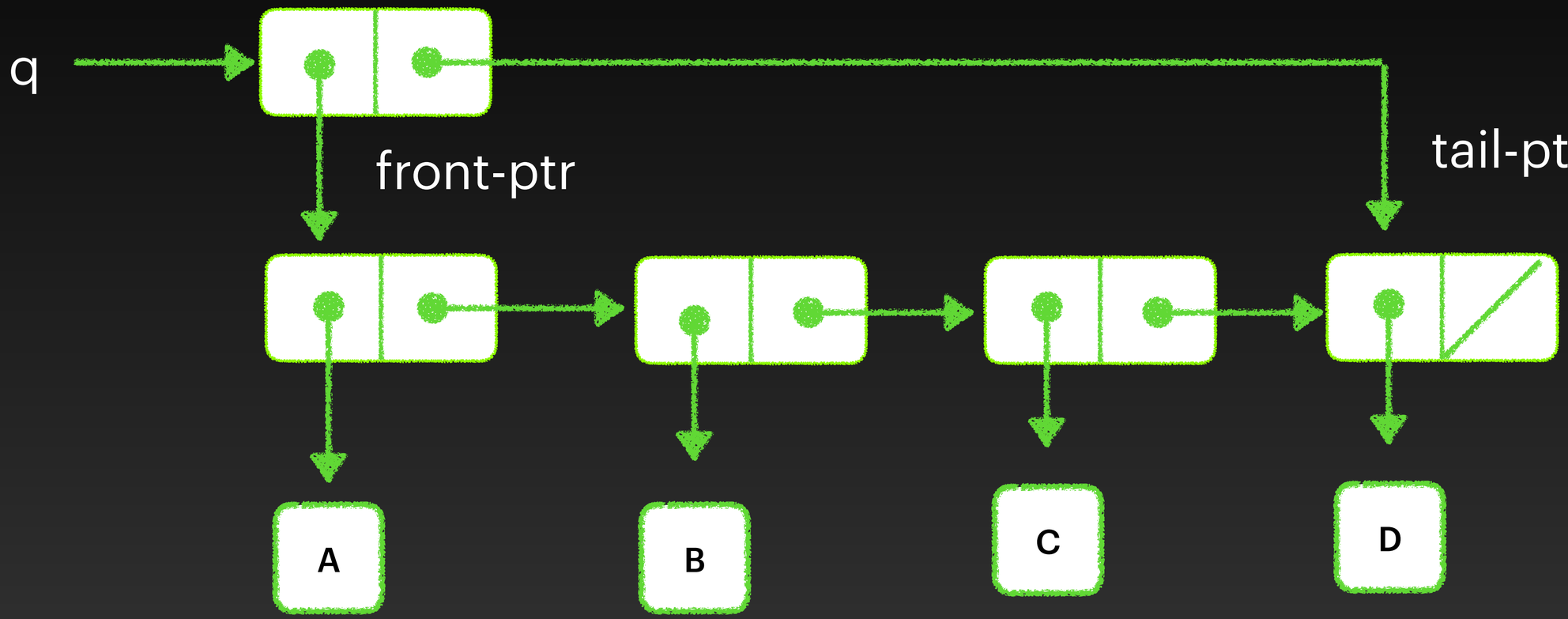
- A list representation of a queue seems to make sense, as the frontmost element is easily accessible by *car*, and one can walk the list by *cdr*ing down the list
- However, this creates a queue with a cost of Θ*n*, increasing in time to execute as the list grows
- Can we do better?

Pair representation



Pair representation

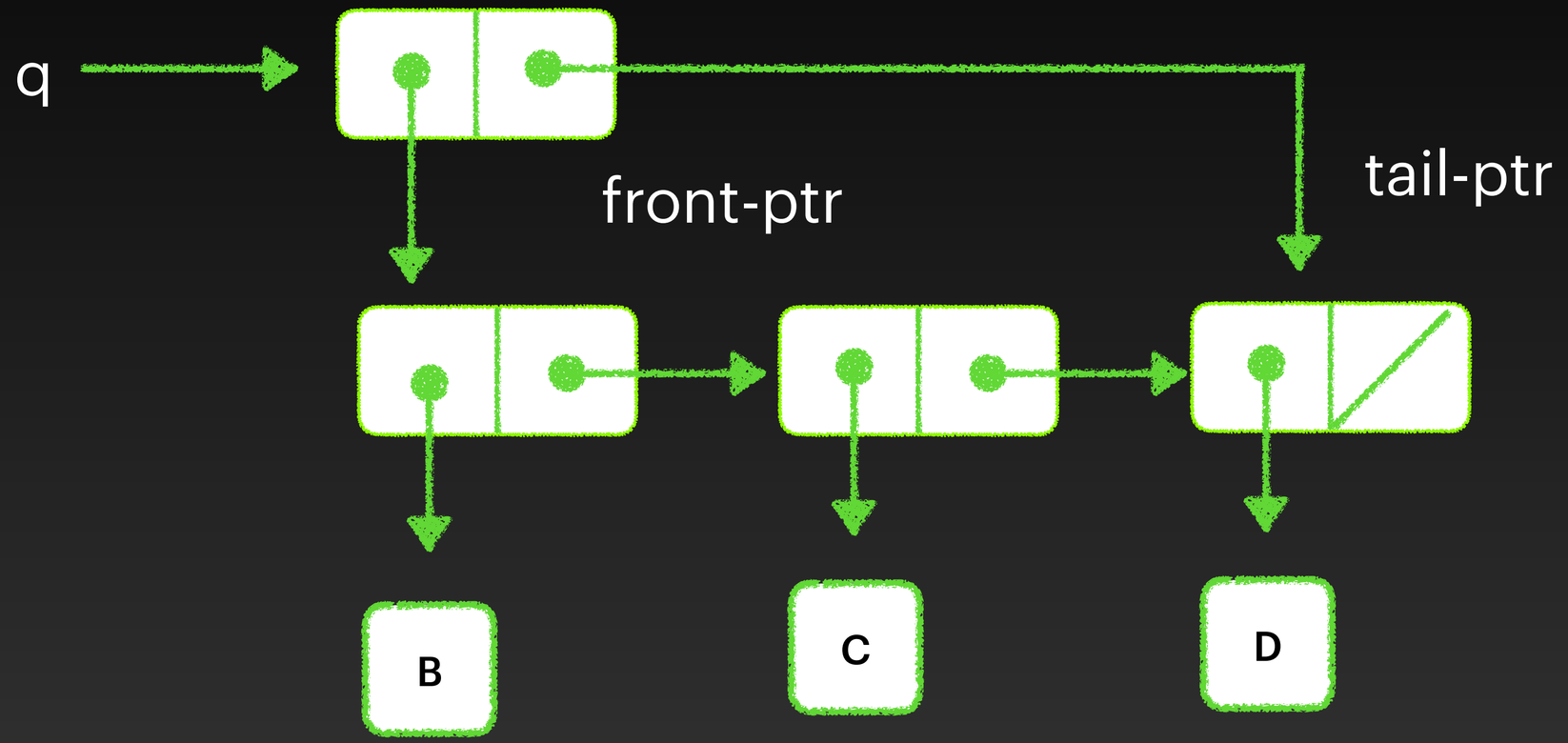
Adding an element



By using a pair of references, our performance can become Θ1!

Pair representation

Removing an element

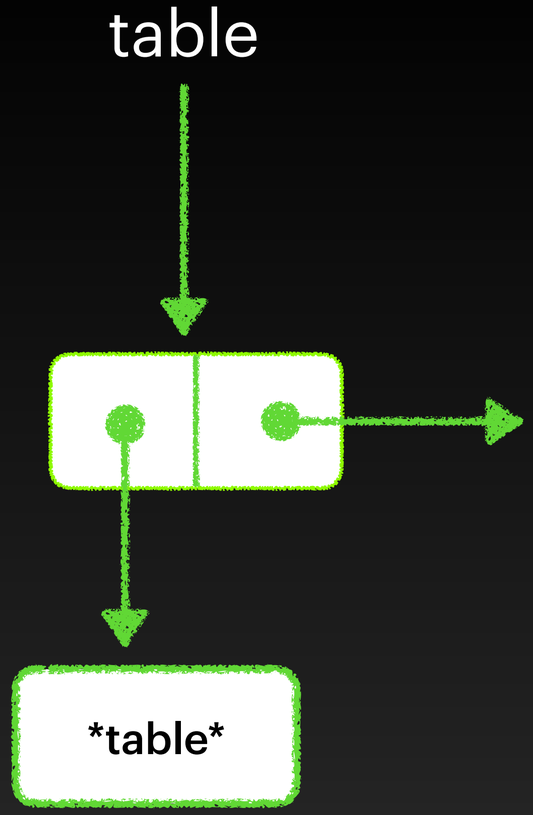


Has the same cost for either implementation

Designing a table

A simple table
single-dimension

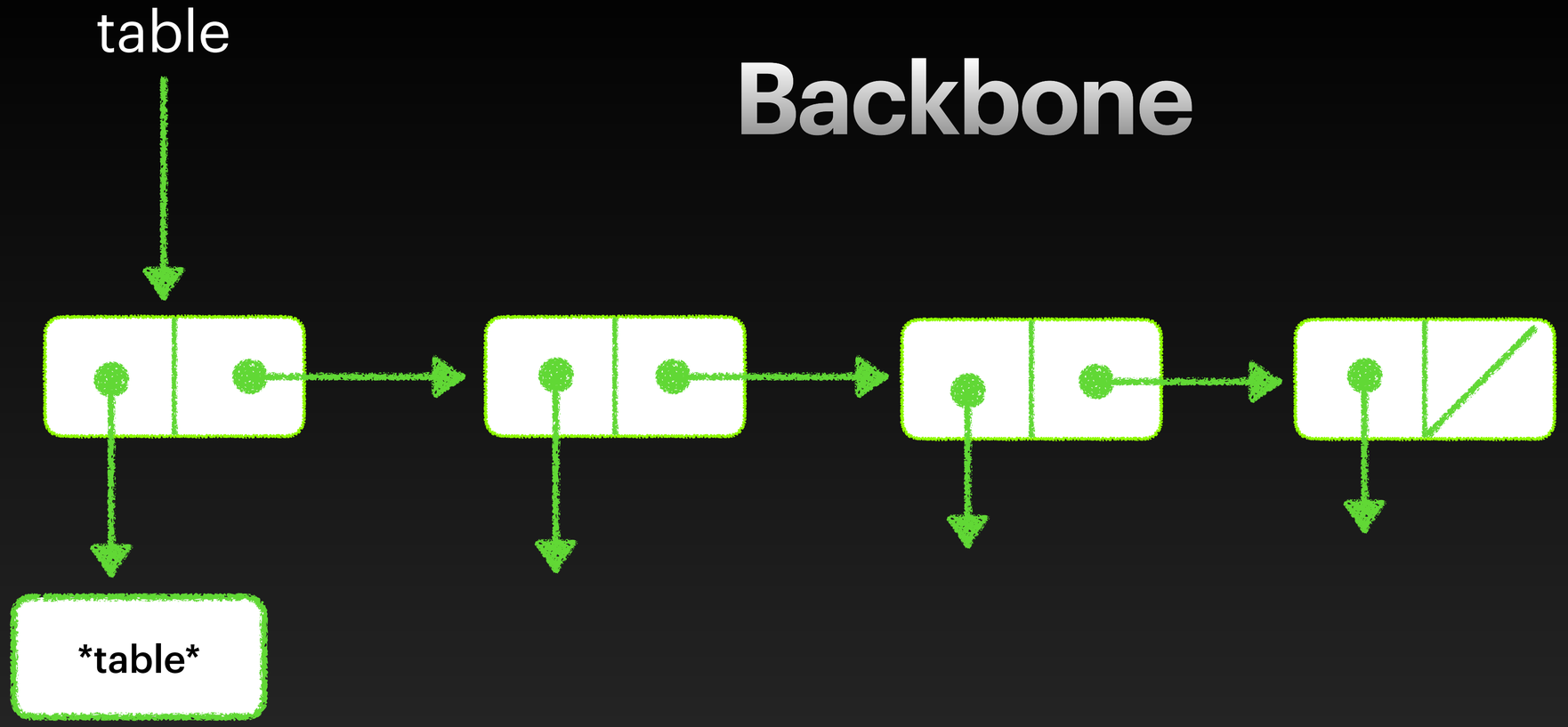
- A table’s primary element type is a 'headed list'
  - A headed list has a special pair at the top containing arbitrary info, plus a pointer to the backbone list
  - The backbone list points to a sequence of pairs containing keys and data



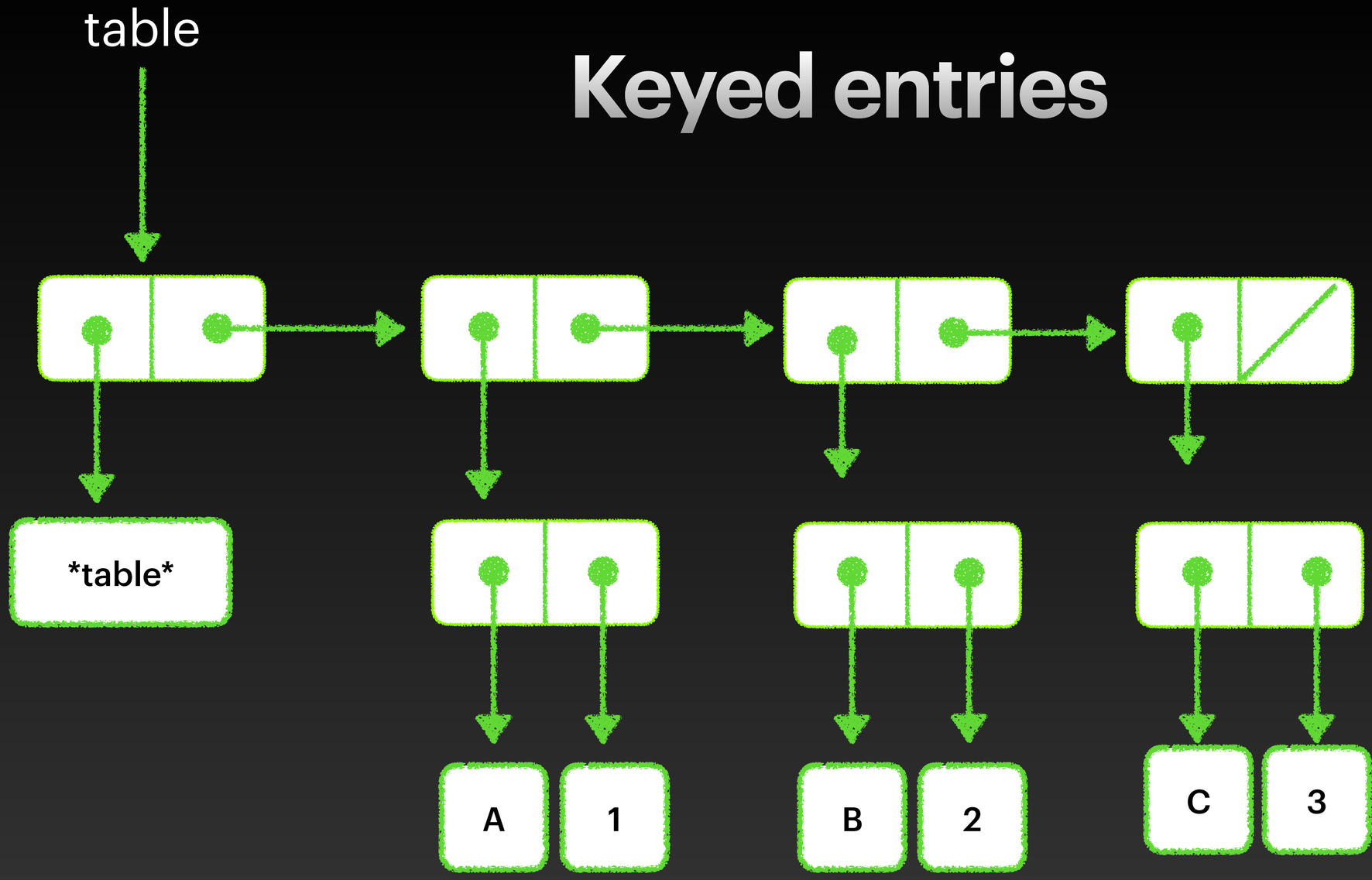
Headed pair

Aribitrary symbol *table* for head of table

Backbone



Keyed entries



a: 1

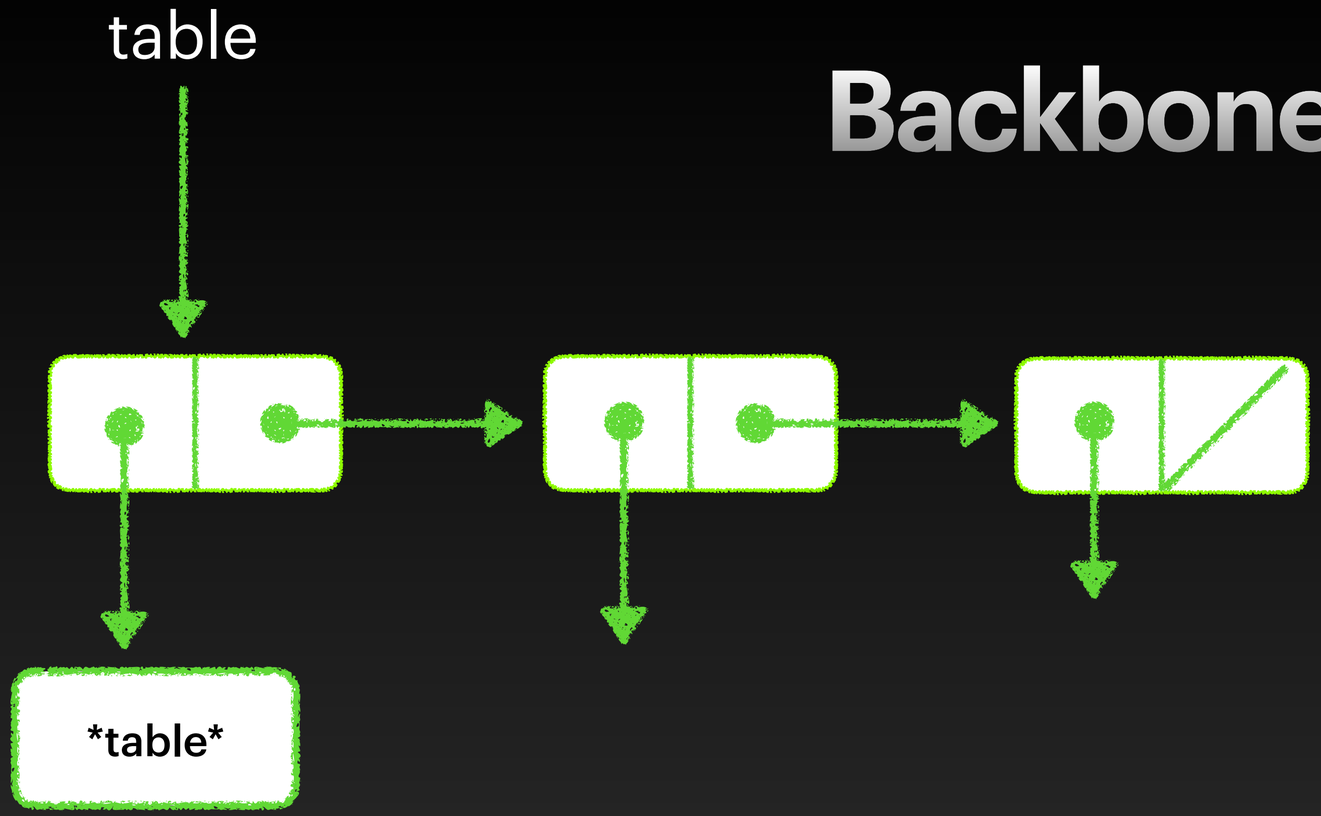
b: 2

c: 3

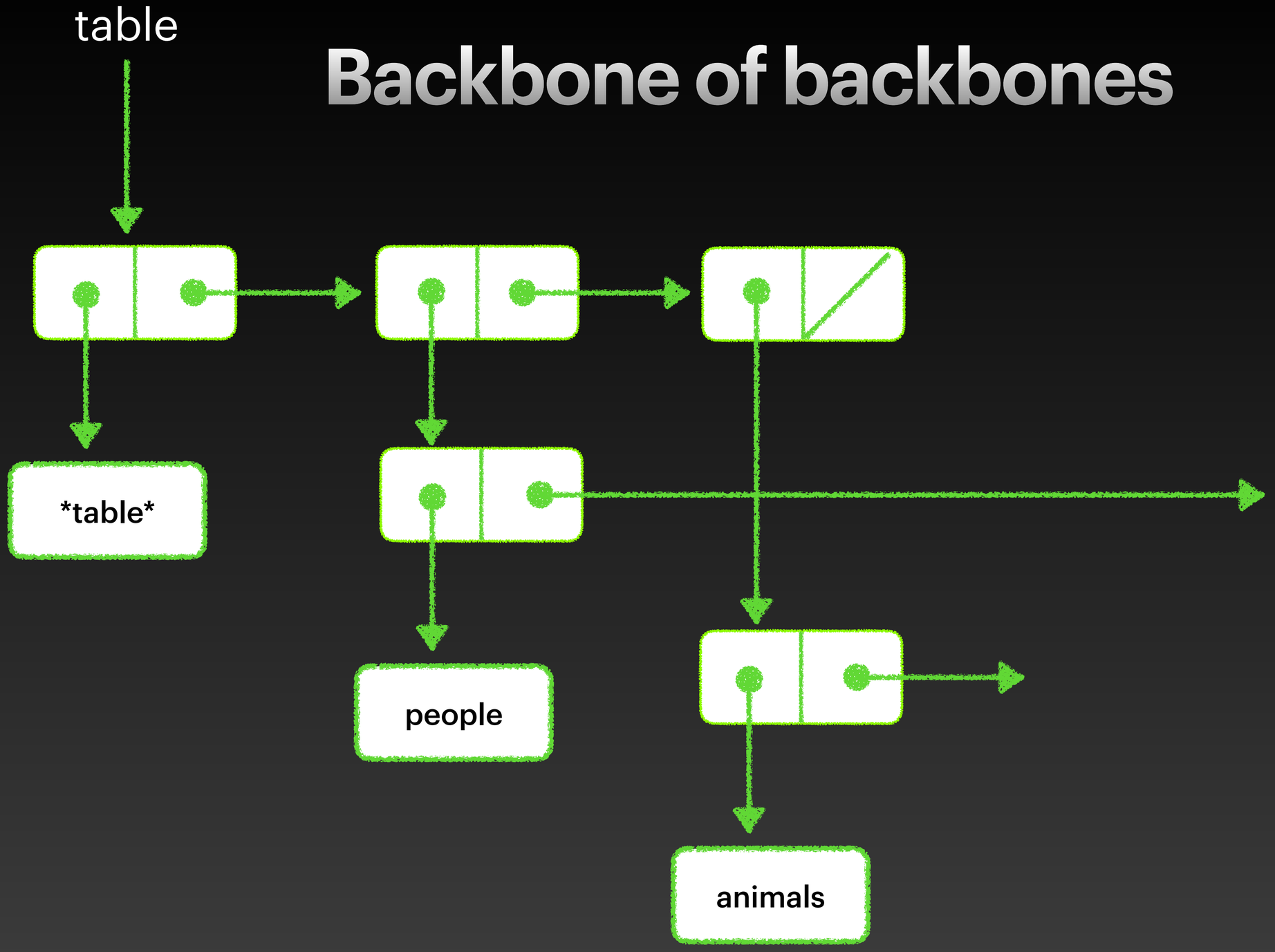
Two-dimensional tables
just as easy

- Making a two-dimensional table is just as easy
- Backbone is a list of backbones
- (recursion strikes again)
- Subtables don’t need special header symbol, just use key for subtable

Backbone



Backbone of backbones



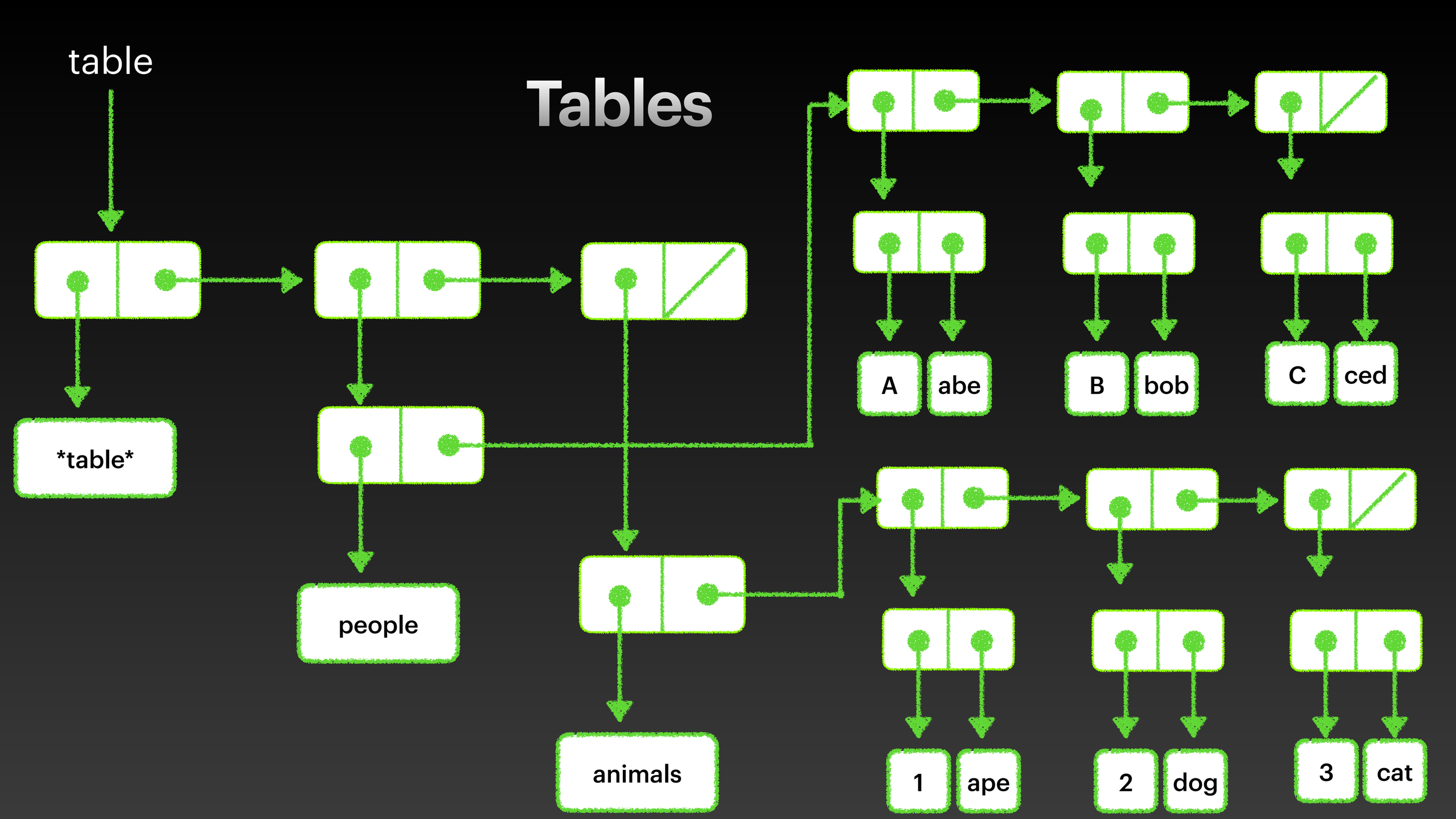


table
Tables
table
people
animals
A
abe
B
bob
C
ced
1
ape
2
dog
3
cat

Homework

Read and draw

- Read section 3.3.1 (at least) of SICP
- Complete exercise 3.15
  - This is drawing a diagram
- Send your drawing as an attachment to an email
- Due date: Tuesday 4/28 no later than 11pm
- Readability counts, perfection doesn’t