Cons and memory

Currying / higher order functions

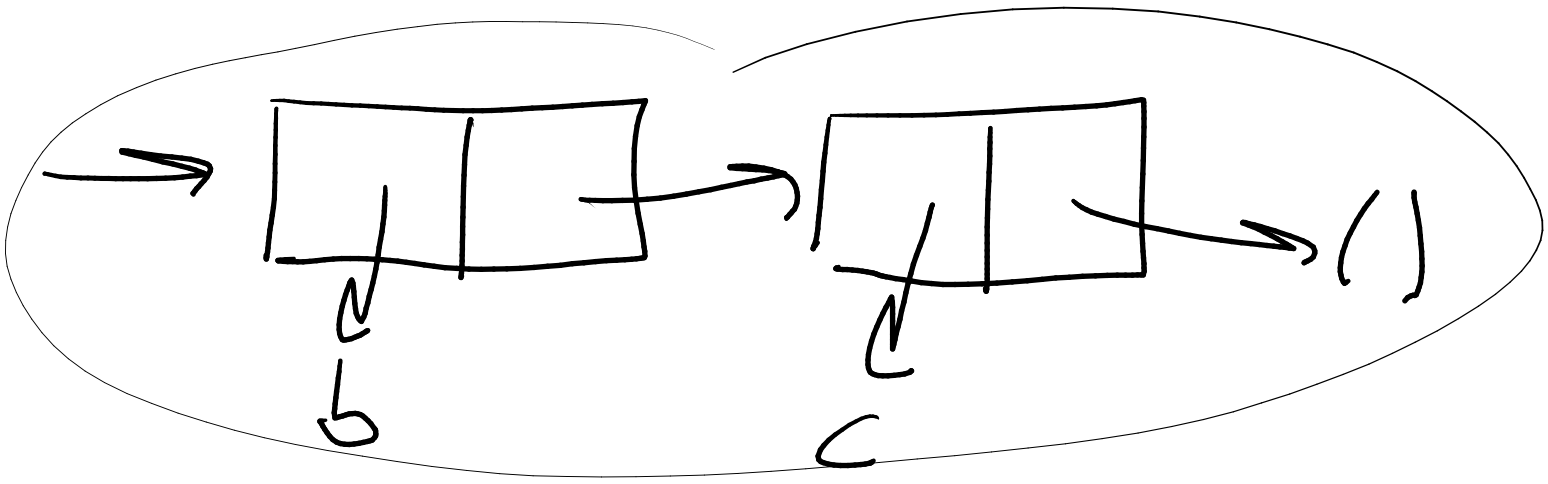
→ (cons 'a '(b c))

↳ (a b c)

(cons '(d e) '(b c))

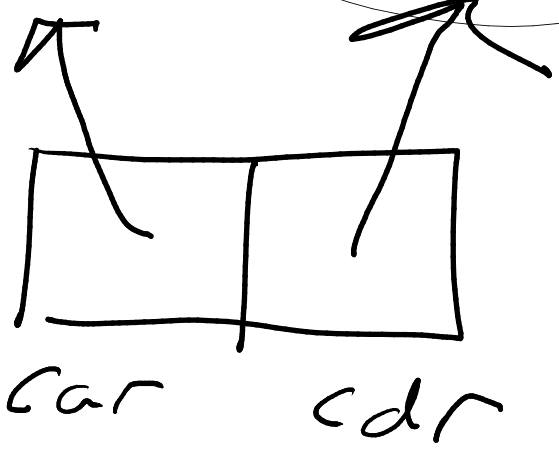
↳ ((d e) b c)

this is all just a result of what cons really does, which is to make a cons cell, aka pair

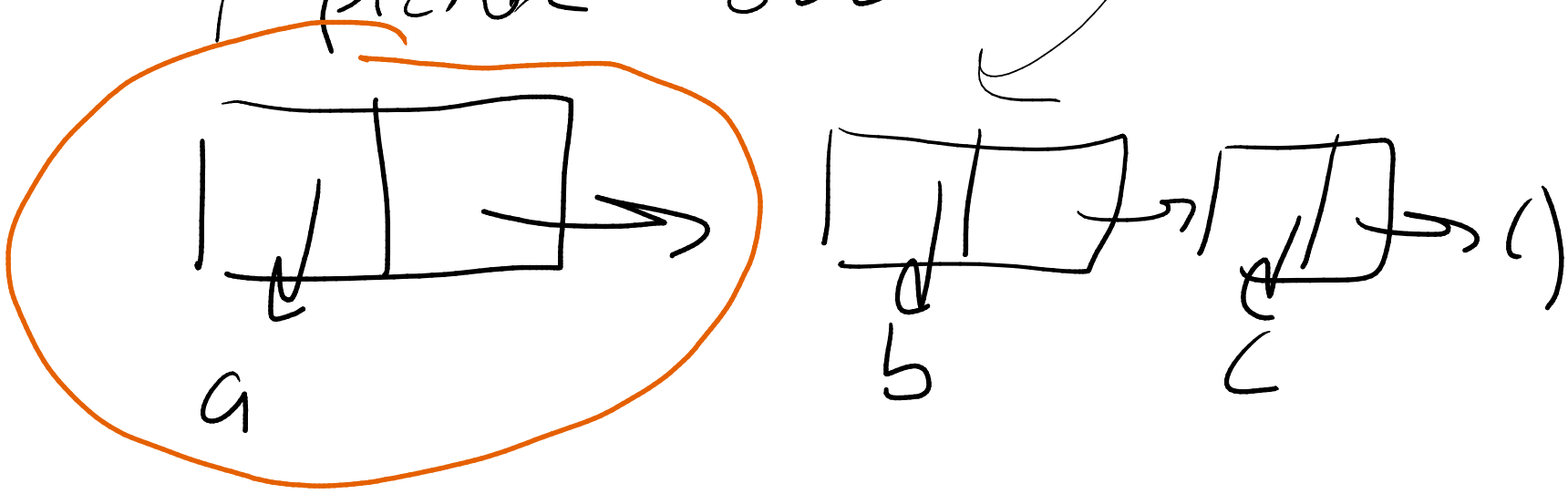


⇒ a

(cons 'a '(b c))

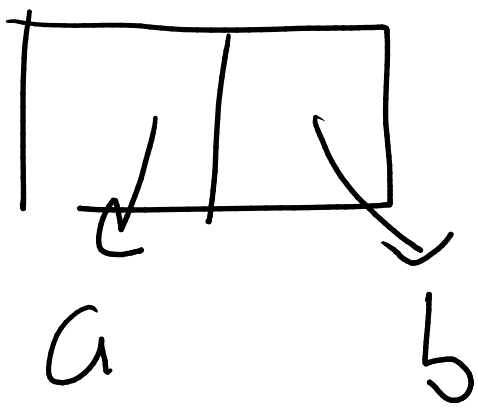


flip picture over



(a b c)

(cons 'a 'b)

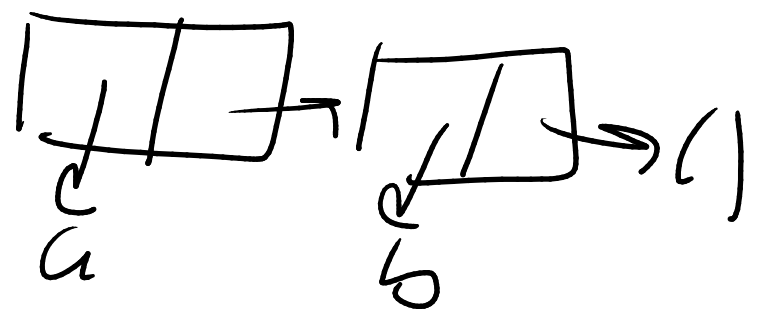


<---- this is not a well formed list, because second ptr is not to a list

(a . b)

how Scheme display

'(a b)

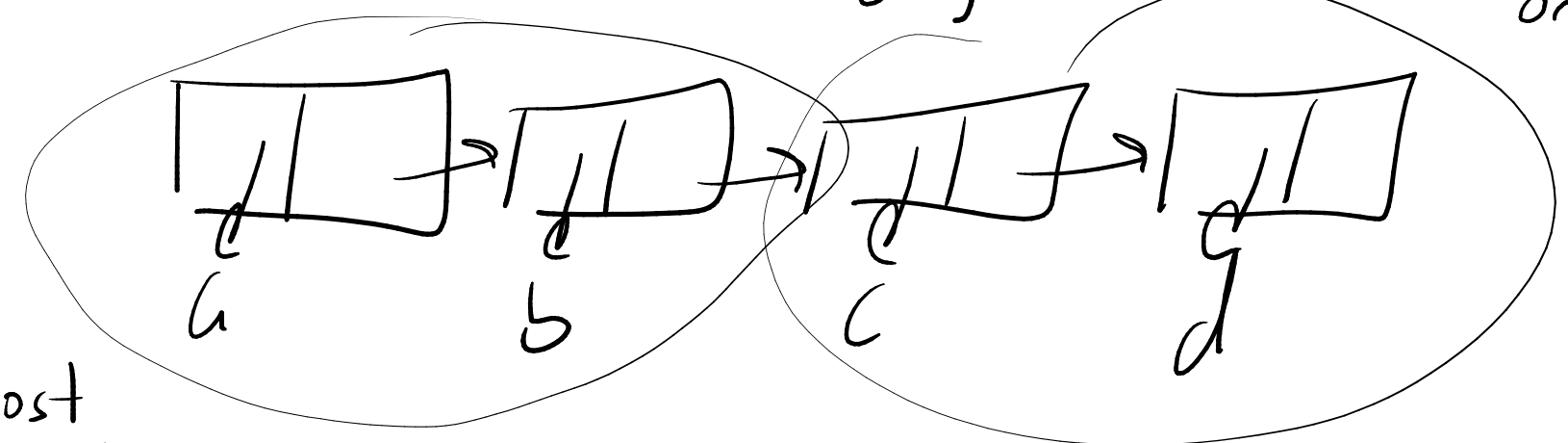


Internally, this is

(a . (b . ()))

(append '(a b) '(c d))

↳ (a b c d)



almost first list, but last ptr changed

Lots of ways of using functions as parts of other functions - or treating functions as data

Currying (Haskell Curry) - an alternative approach to having multiple parameters for a function

---

"Normal" parameters

```
(define mult
  (lambda (a b)
    (* a b)))
```

```
(mult 3 5)
↳ 15
```

Currying - make a function of one parameter that returns a function that takes second parameter

```
(define mult
  (lambda (a)
    (lambda (b)
      (* a b))))
```

The first fuction I call returns a second that I can use multiple times if I want

Useful scenario: lots of work needed on first param, and not on the second.

Without this, every time you call the function you would redo all of the work on the fist parameter

Higher order functions
- one function takes another as a parameter

(These ideas are used all over the place in parallel computing)

(fold-left f 'a '(b c d))

f — function of 2 params
'a — starting value
'(b c d) — list

(f (f (f 'a 'b) 'c) 'd)

(f 'a 'b)

↓

(f 'c)

↓

(f 'd)

(fold-left + 0 '(1 2 3))

(+ 0 1) → 1

(+ 1 2) → 3

(+ 3 3) → 6

(fold-left / 24.0 '(1 2 3))

(/ 24.0 2) → 12.0

(/ 12.0 3) → 4.0

✓