How do lists really work in Scheme?

(a b c)   list

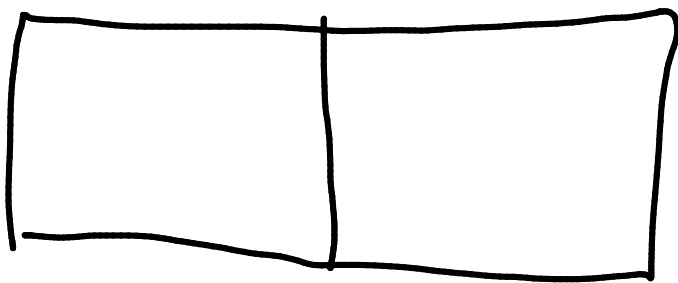stored as a linked list

empty list
"null" ←
"None"

a   b   c
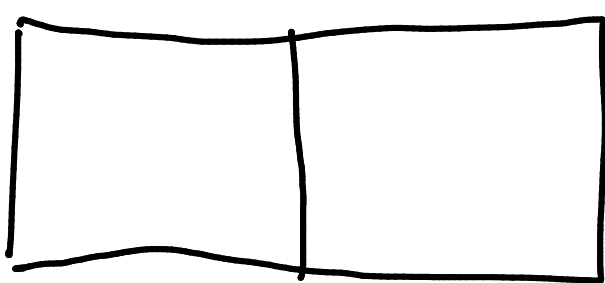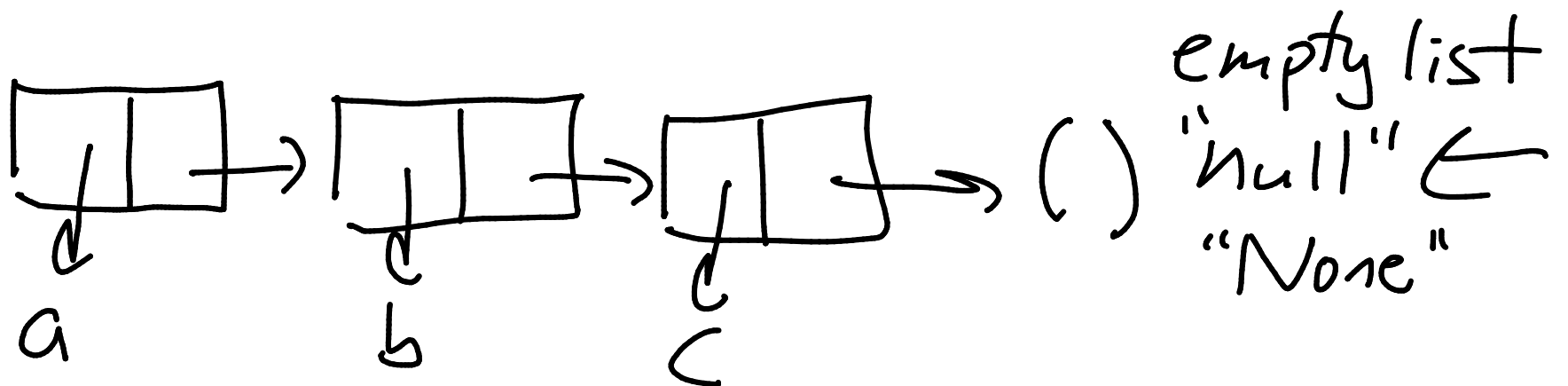
"pair"
"cons cell"
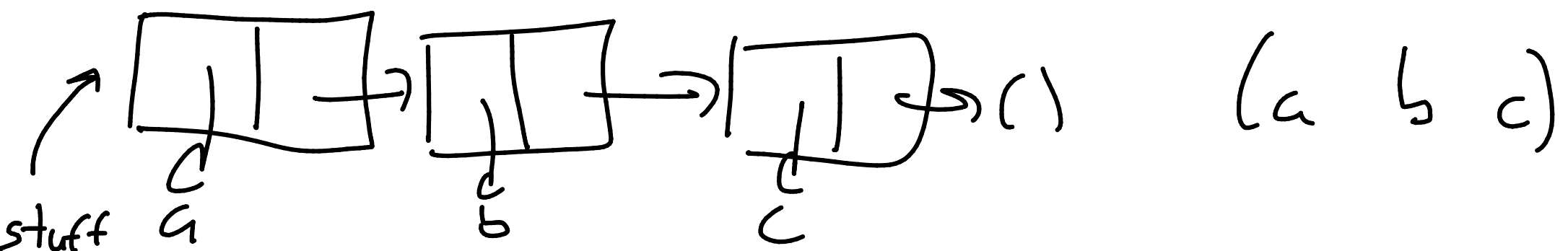"cell"

history
1957 John McCarthy MIT
 -created Lisp (effectively Scheme)

address | decrement
register | register

made use
of registers
-address register
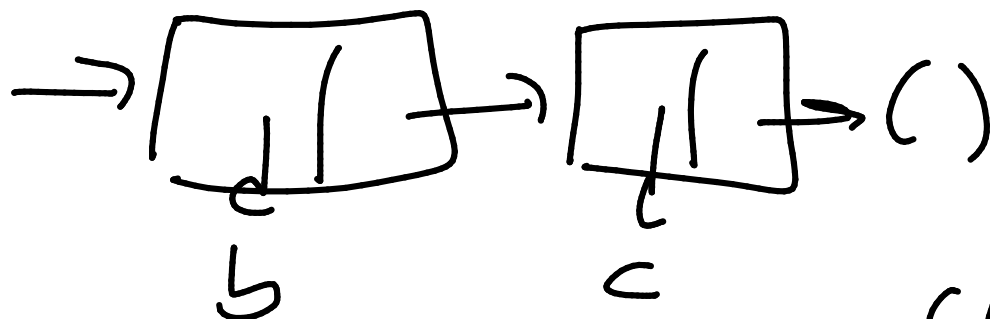-decrement register

(a b c)

first item in stuff
- "contents of address register"

  (car stuff)

- rest of list

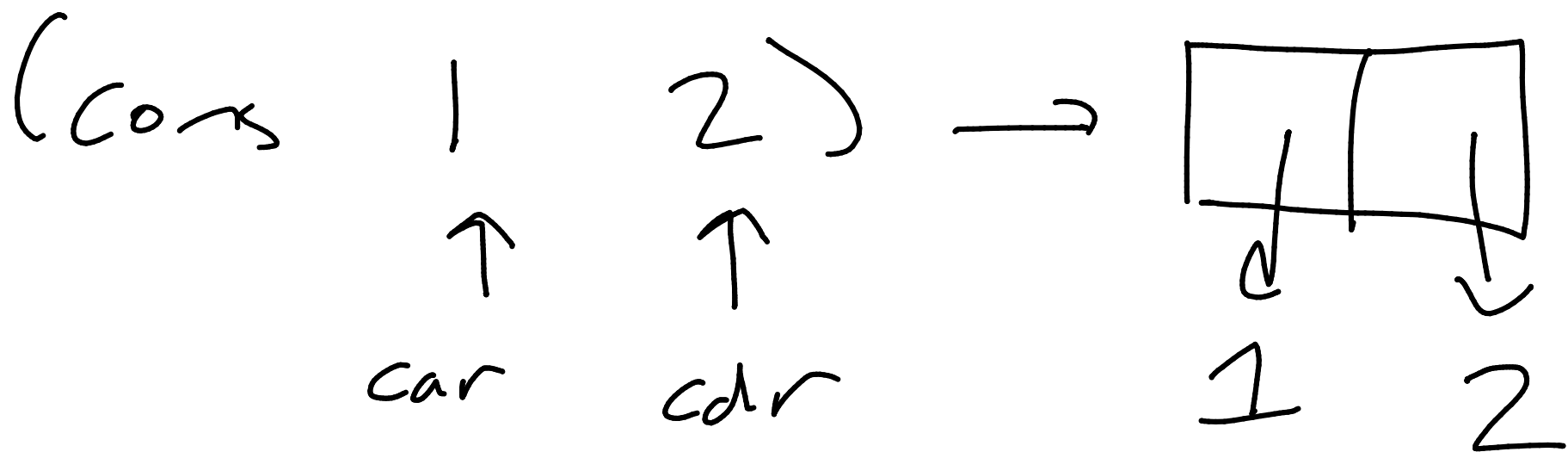
(b c)

  - "contents of decrement register"

  (cdr stuff)

List don't have to be flat, or even properly made.

Cons - "stick onto front of a list"   (naive)

  (cons 1 '(2 3)) → (1 2 3)

What cons really does is make
a new cons cell

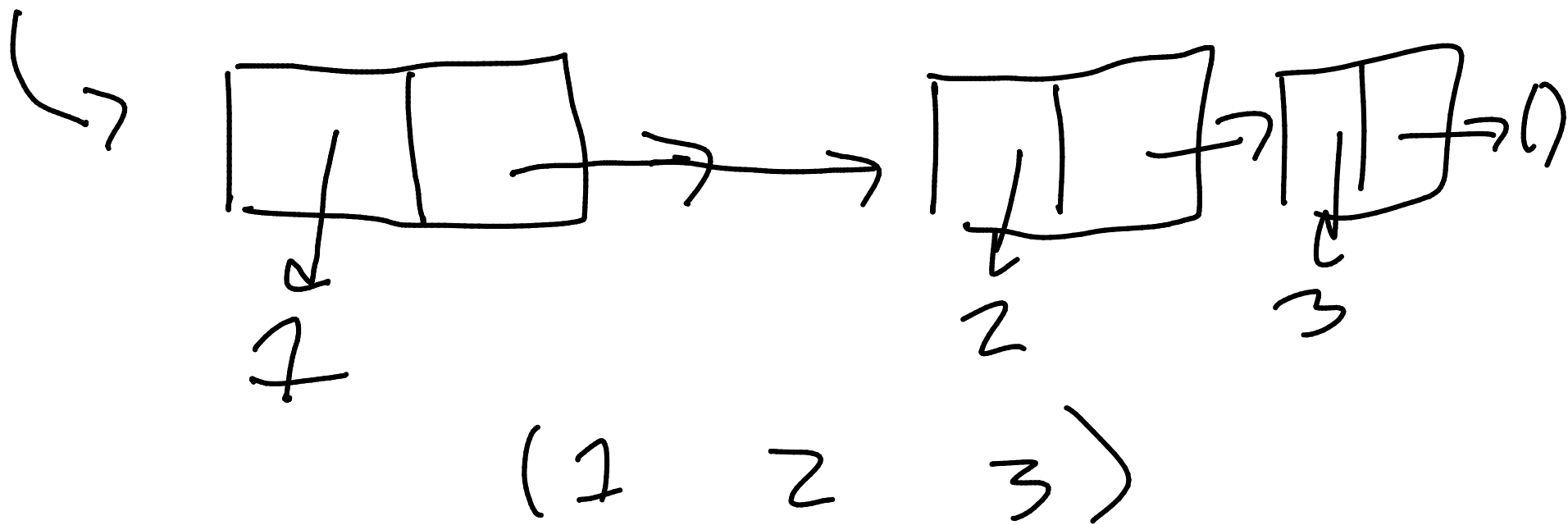(cons 1 2) → [diagram: cons cell pointing to 1 and 2]

↑ car   ↑ cdr

1   2

"improper list"

(1 . 2)

a single pair
with car on
left and cdr
on right

(cons 1 '(2 3))

↳ [diagram: linked list cells with 1, 2, 3 ending in ()]

1   2   3

(1 2 3)

(1 2)

1

2

1

2   ()

$$(1 \cdot (2 \cdot ()))$$
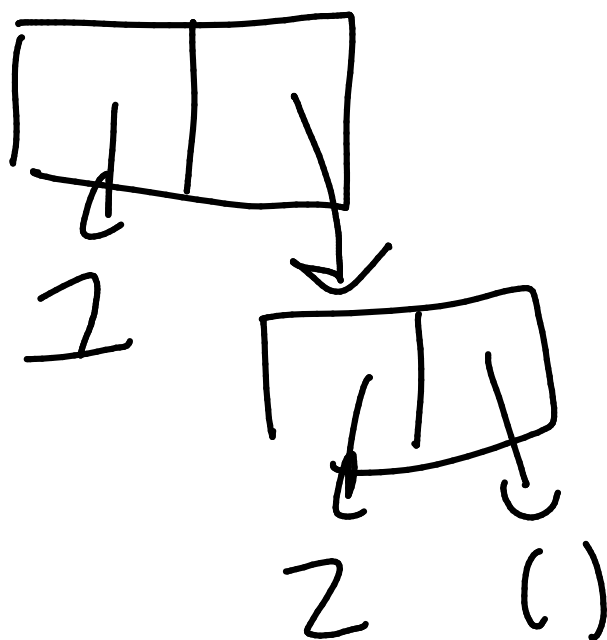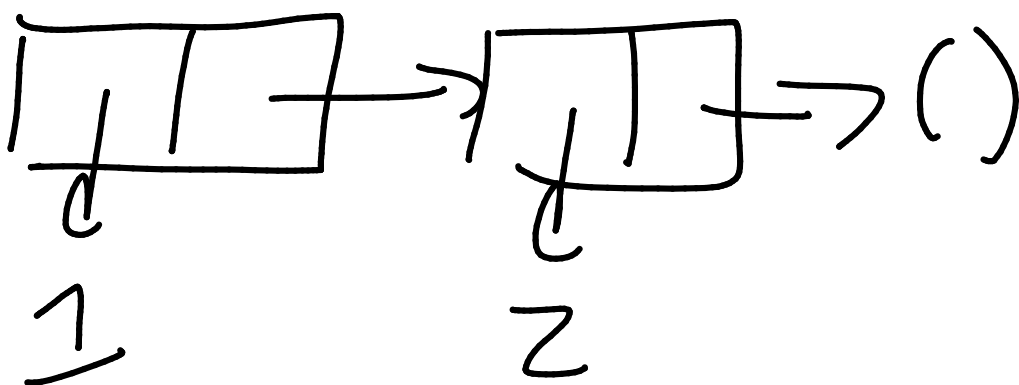
everything above is the same

But (1 2) is _not_ (1 . 2)

(Single quote)

By default, Scheme evaluates everything
you enter.

                    evals
(+ 1 2)   ⟹ 3

(cons 1 2) $\xrightarrow{evals}$ (1 . 2)

2 $\xrightarrow{evals}$ 2   any constant
                         evals to itself

(1 2 3) $\xrightarrow{evals}$ error that
                             1 is not
                             a function

'(1 2 3)
↖ don't eval it

'(1 2 (+ 3 5))
  ⟹ (1 2 (+ 3 5))

Cons - makes a cons cell

- common application - make a new
  list with add'l item on front

(cons 1 '(2 3)) ⟹ (1 2 3)
       ‿   ‿not a list
       2 prams

```
(append '(1 2) '(3 4))        ← list
                        2 params
  ⇒ (1 2 3 4)

(list 1 2 3 4 5)
        many params
  ⇒ (1 2 3 4 5)
```

```
(cons '(1) '(2 3))
  ⇒ ((1) 2 3)

(list '(1) 2 '(3) '(4 5))
  ((1) 2 (3) (4 5))
```

Creating functions

In Scheme, there is a built-in function that creates functions

⟶ <u>lambda</u>

1930s Alonzo Church designed a prog lang based on nothing but functions

```
(lambda (arguments)
    body)
```

Create a function to add 2 numbers

```
(lambda (x y)
    (+ x y))
```

```
(define my-add
    (lambda (x y)
        (+ x y)))
```

(my-add 3 5)

·