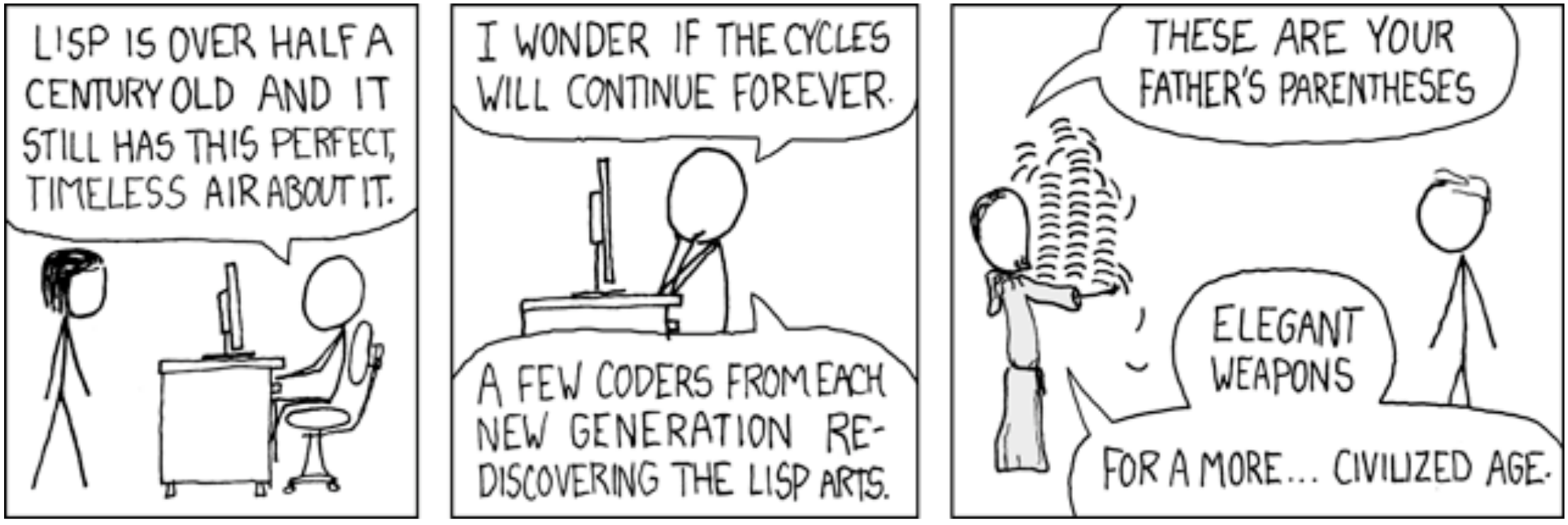
Datastructures in Racket

Part 1


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.

For today's class, we're going to build every data structure out of three things

The first is **atoms**

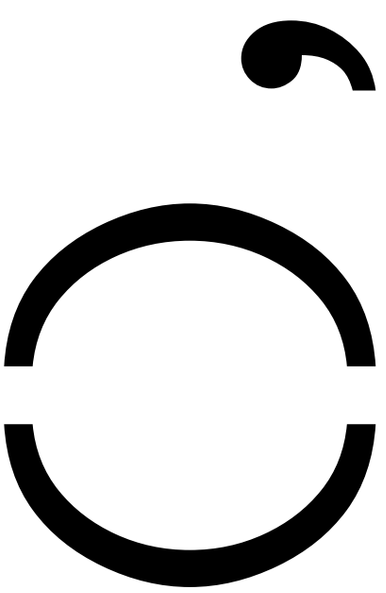
These are the primitive things in the language

‘symbol

1

These are like “`int`” and “`char`” in C++

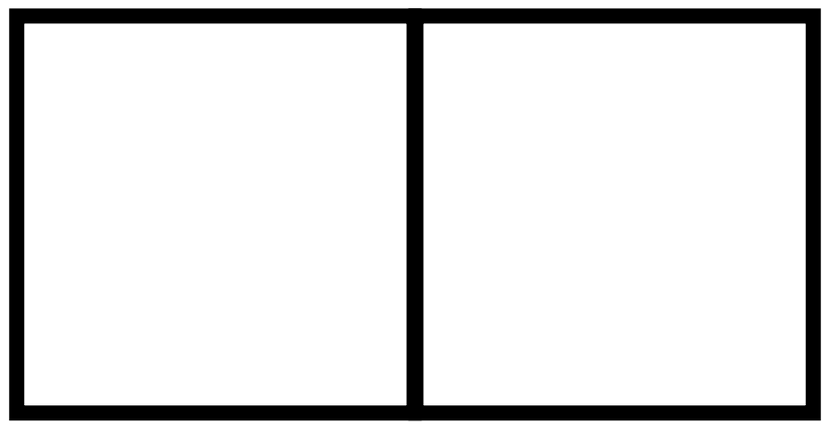
The second is the **empty** list



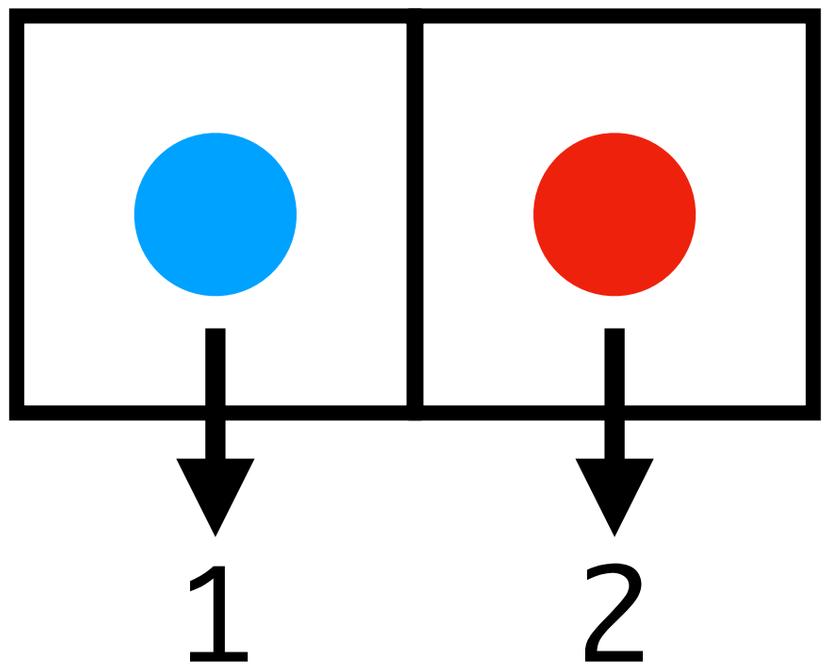
The last is **cons**

Cons is a function that takes two values and makes a pair

That pair is represented as a **cons cell**



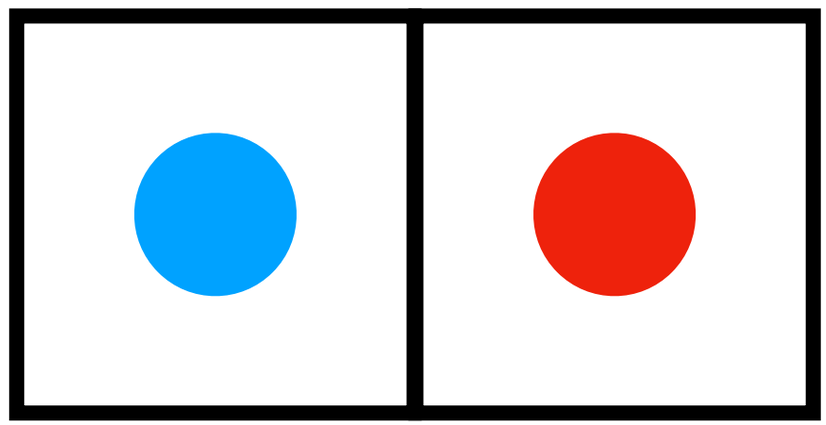
(cons 1 2)



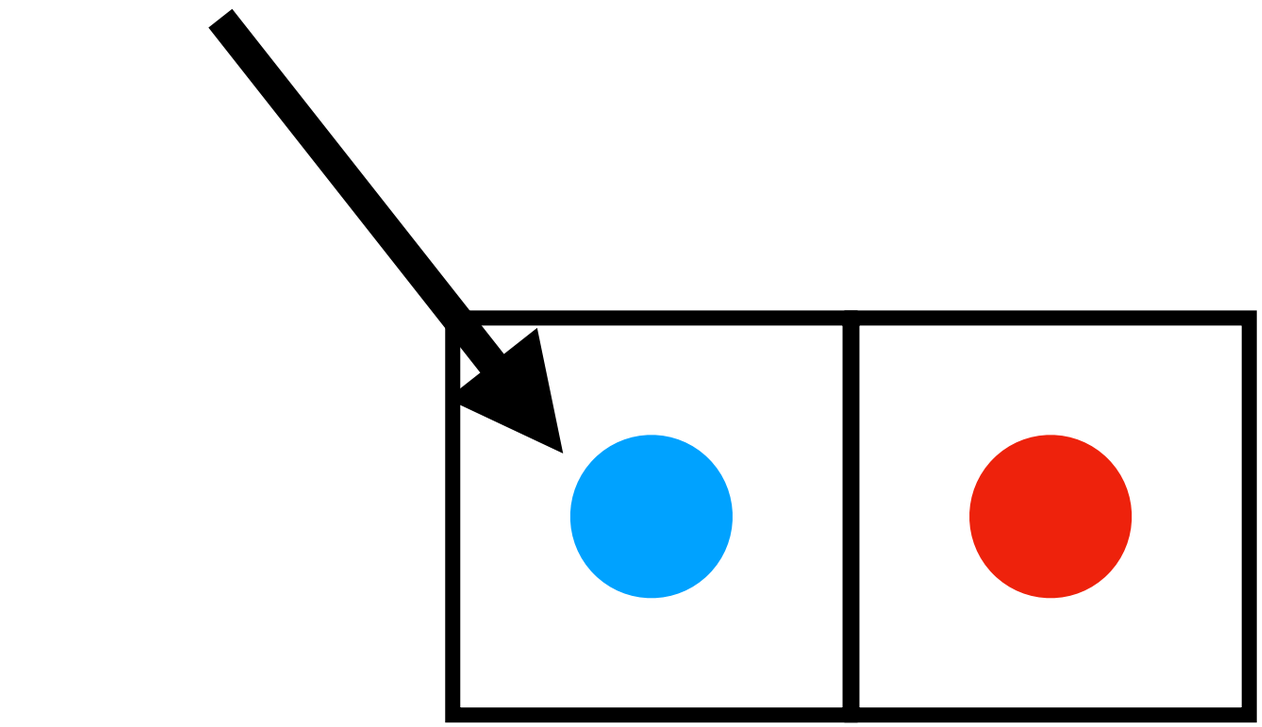
cons is the the natural

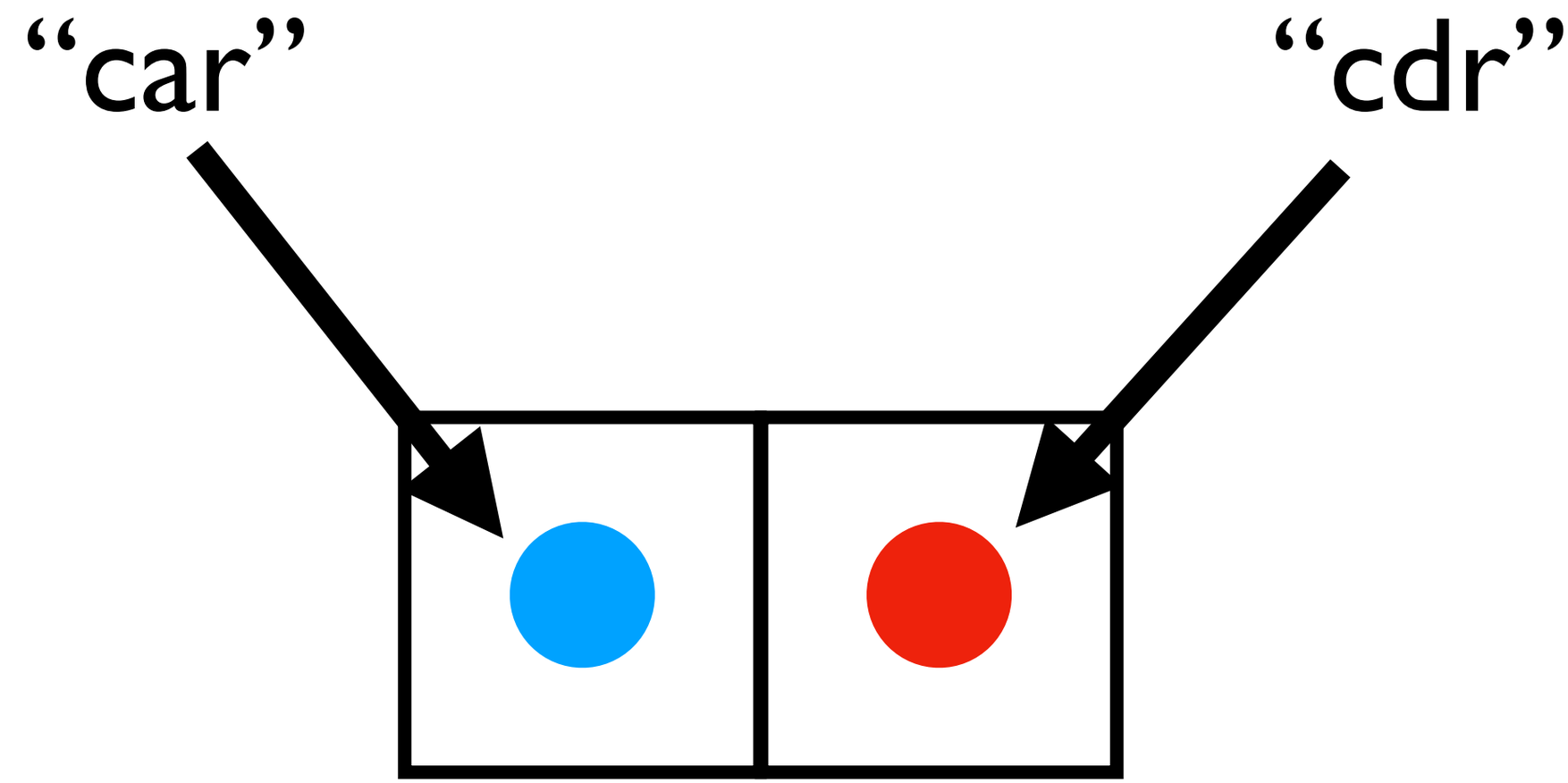
constructor of the language

I use two strange words to refer to the elements of this cons cell



“car”




“car”
“cdr”

Because car and cdr break apart what I build with cons, I call them my **destructors**

And that’s all

And that's all

Atoms	‘sym 23 #\c
Empty list	‘()
cons	(cons ‘sym 23)
car/cdr	(car (cons ‘sym 23))

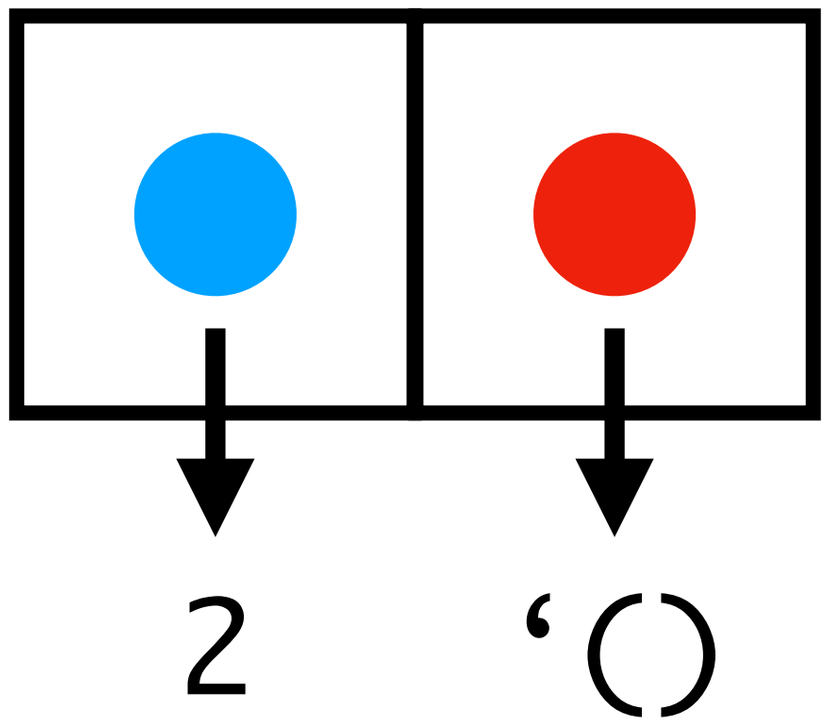
Using just this, I can make a list

Using just this, I can make a list

(And everything else in the world, but we'll get back to that…)

If I want to make the list containing 2 1 do this

(cons 2 '())



When I do this, Racket prints it out as a list

‘(2)

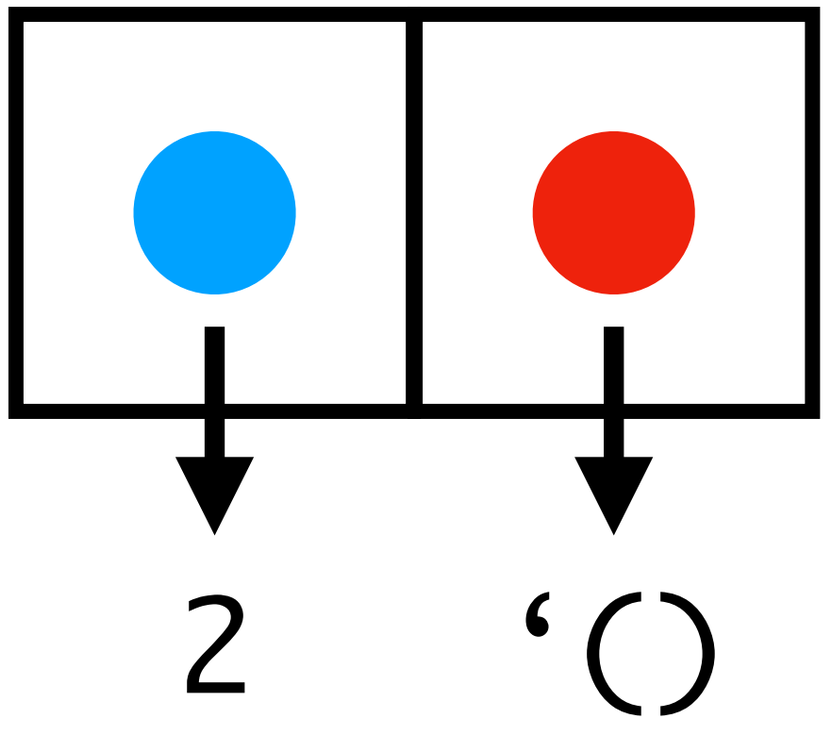
The way to read this is

"The list containing 2, followed by the empty list."

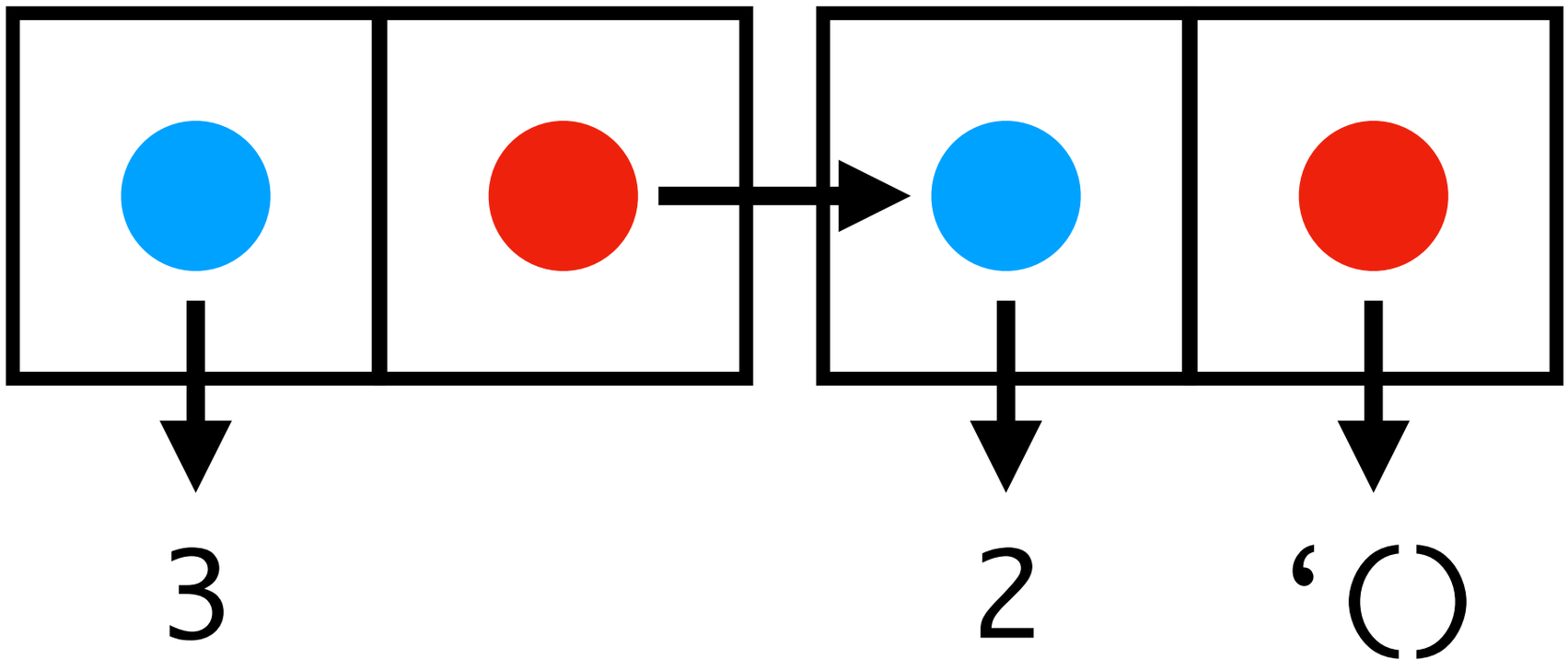
Just as I can build lists of a single element, I can build larger lists from smaller lists…

And I do that by stuffing lists inside other lists…

(cons 2 '())



(cons 3 (cons 2 '()))



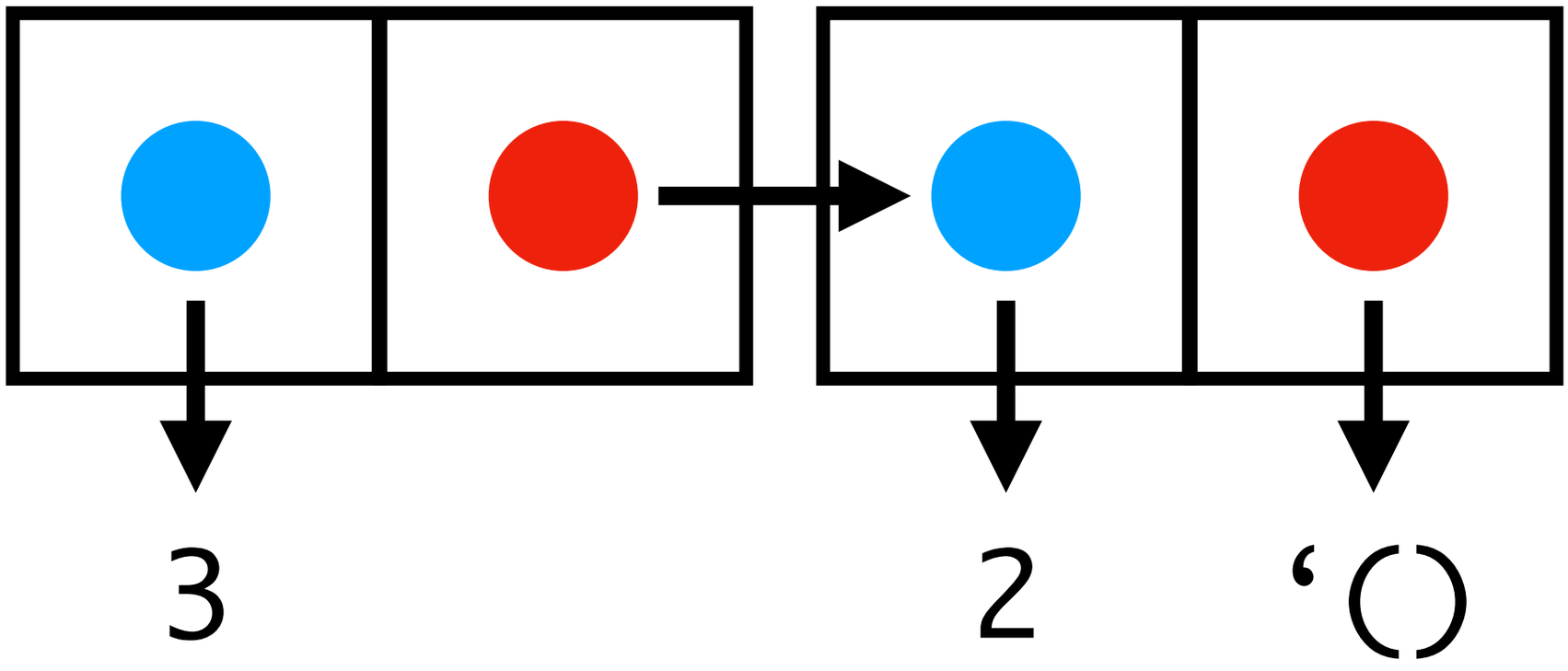
Racket will print this out as

‘(3 2)

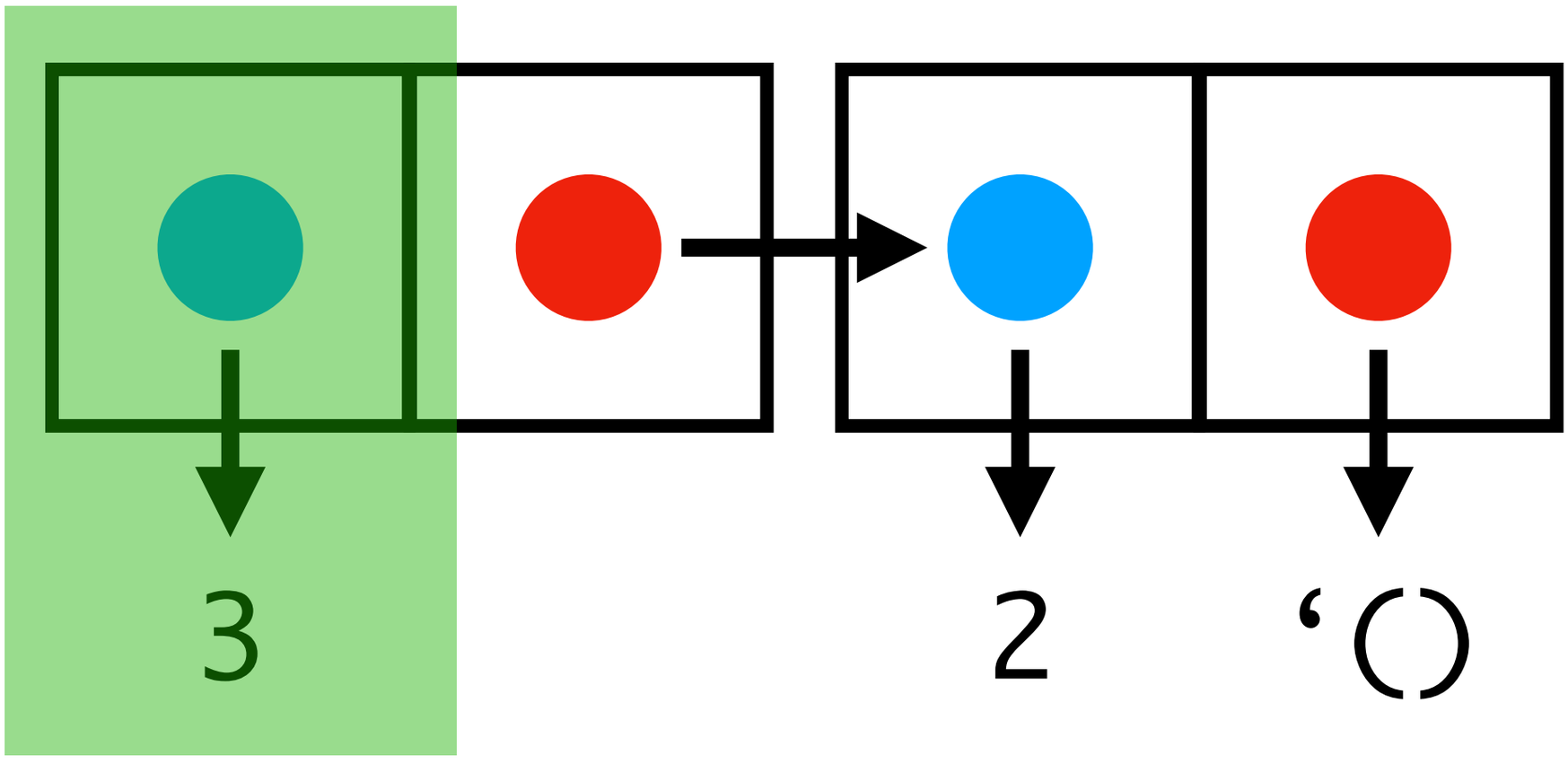
Of course, I probably need at least numbers as primitives right?

To get the head of a list, I use `car`

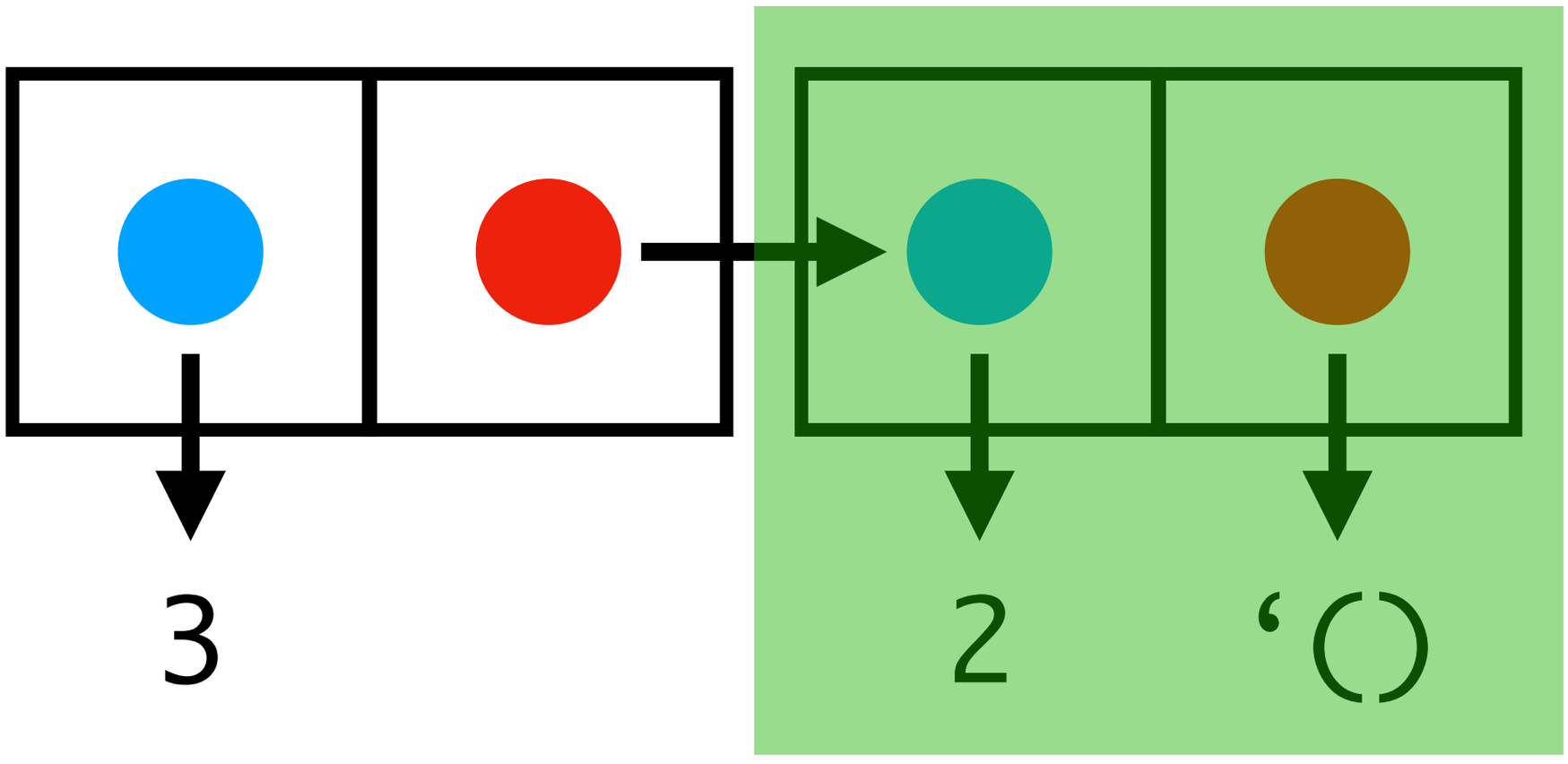
(cons 3 (cons 2 '()))



```
(car
   (cons 3 (cons 2 '())))
```

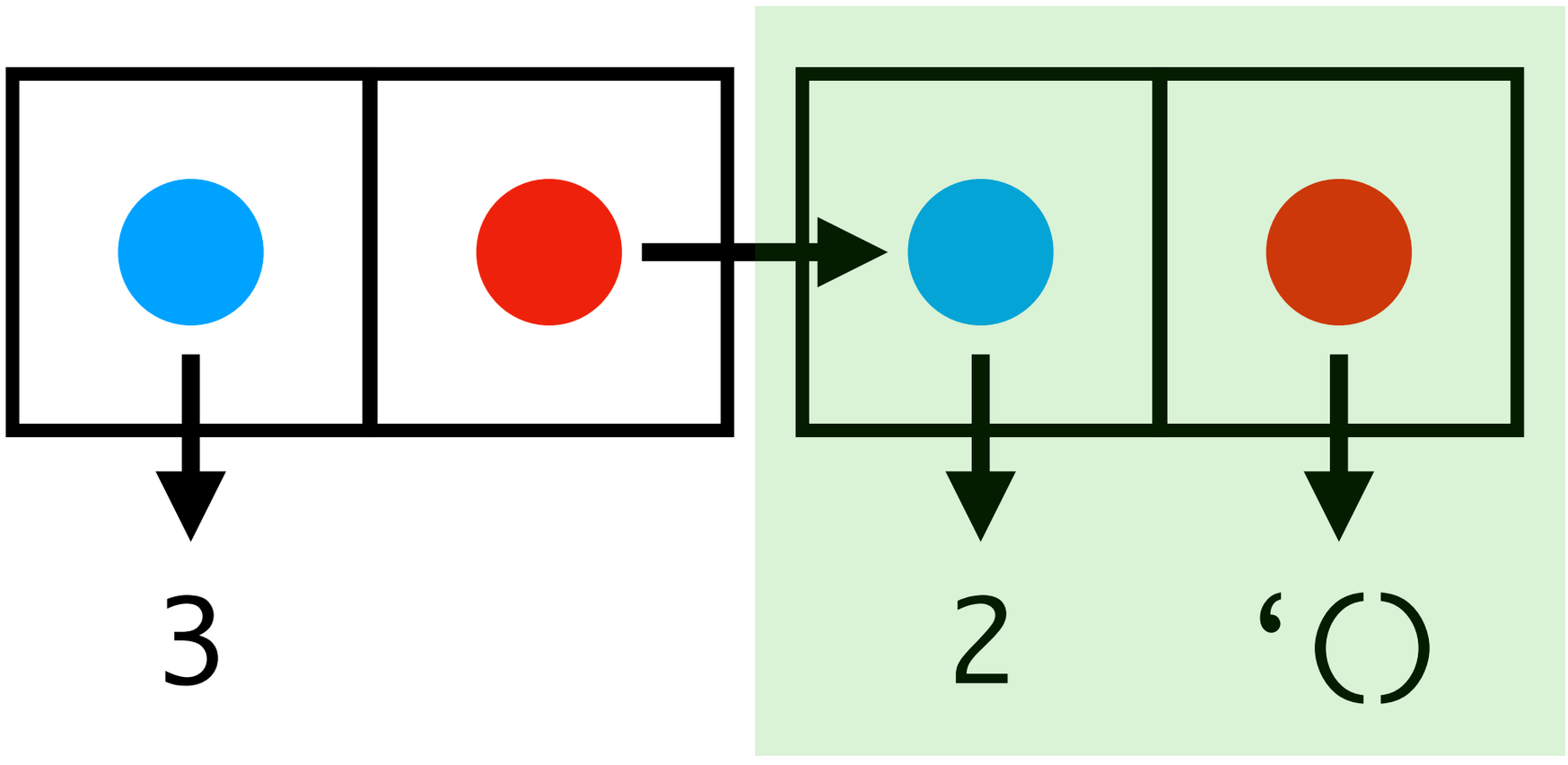


```
(cdr
   (cons 3 (cons 2 '())))
```

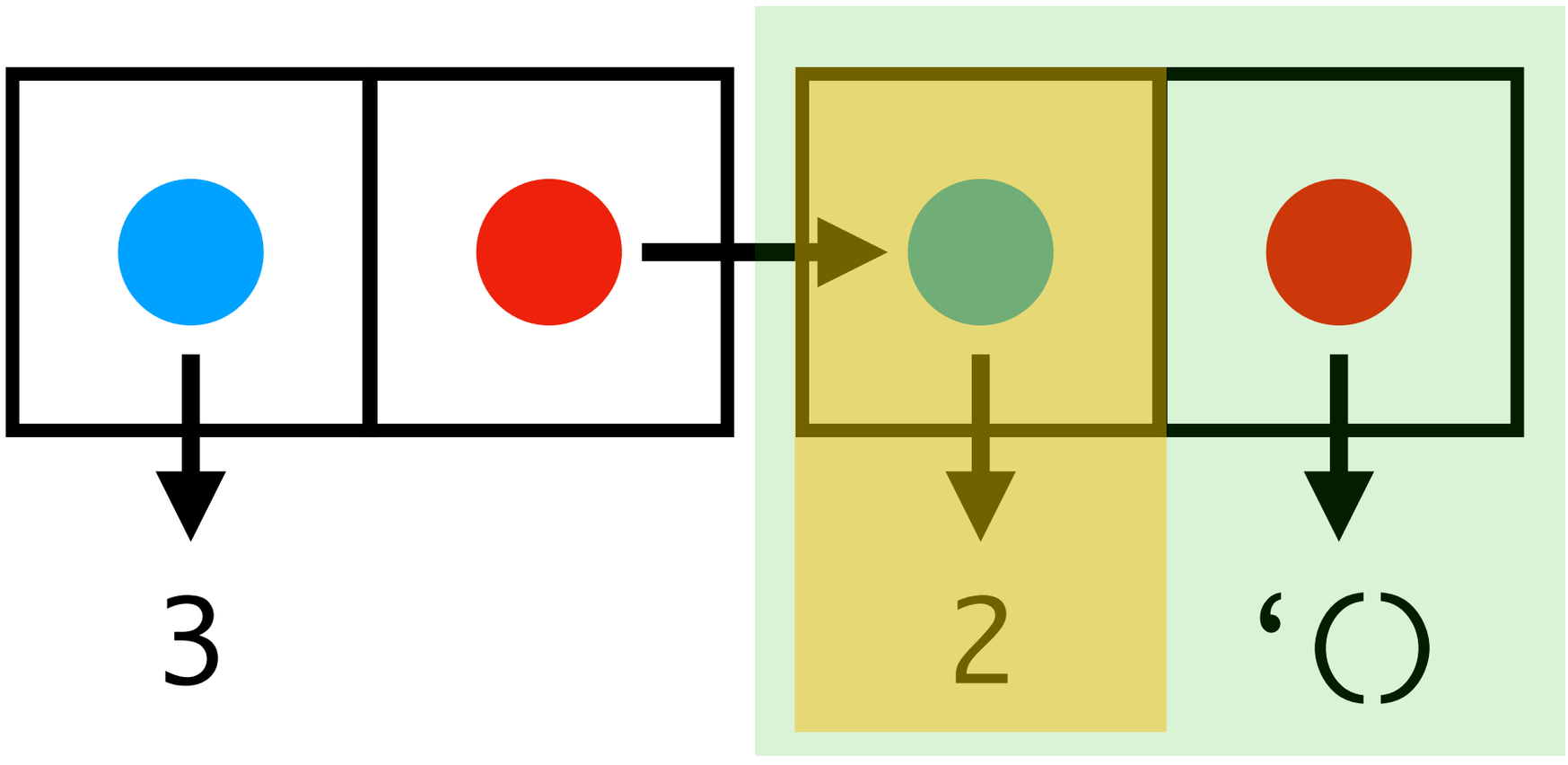


So now how would I get the second element?

```
(cdr
   (cons 3 (cons 2 '())))
```



```
(car
 (cdr
    (cons 3 (cons 2 '()))))
```



Racket abbreviates

```
(cons 1 (cons 2 (cons…(cons n '())…)))
```

as…

```
'(1 2 … n)
```

If I wanted to write out lists, I could do so using

```
(cons 1 (cons 2 …))
```

How do I get the nth element of a list?

```
(define (nth list n)
  (if (= 0 n)
      (car list)
      (nth (cdr list) (- n 1))))
```

Now, write `(map f l)`

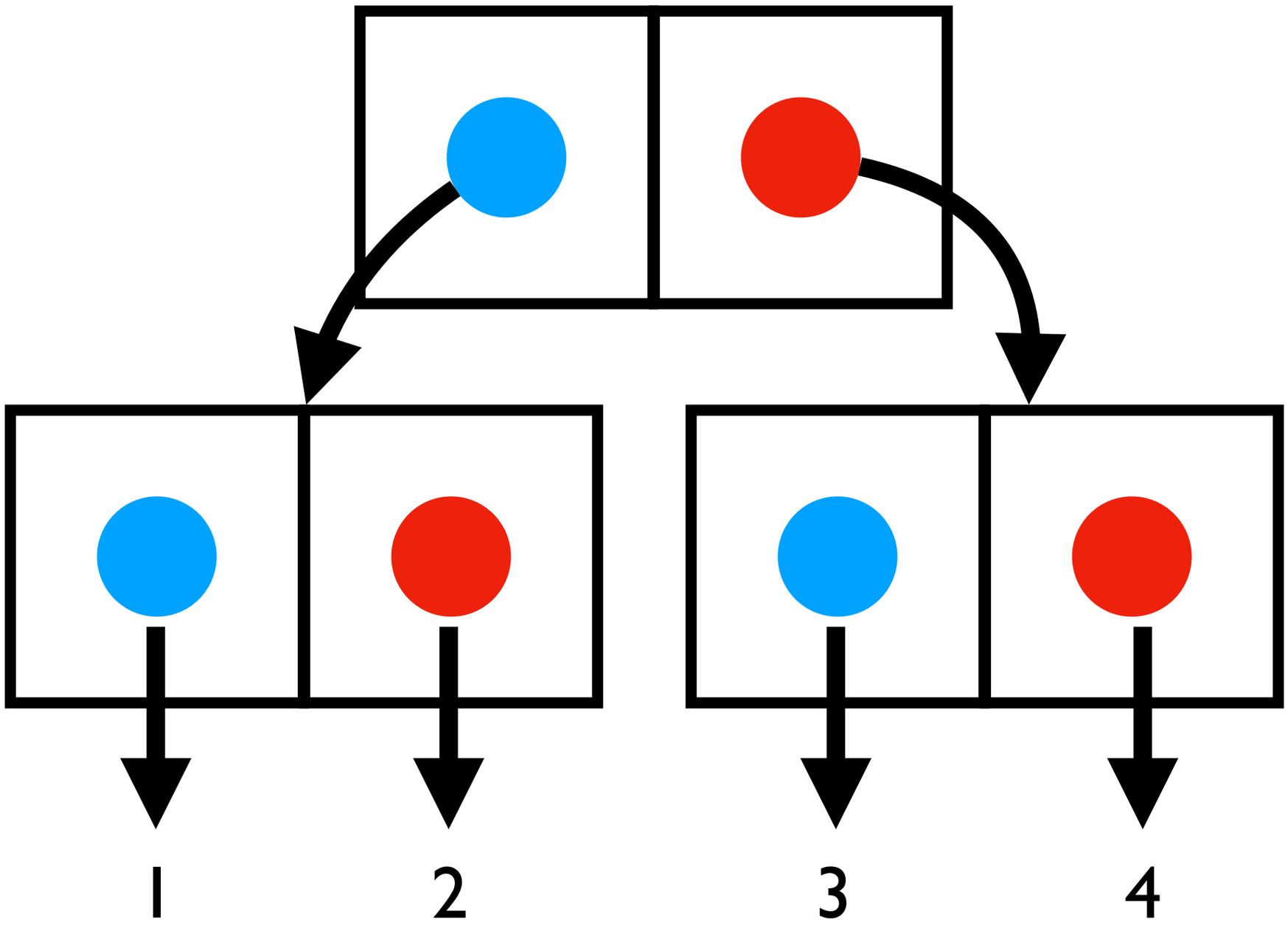
Writing lists would get quite laborious

Instead, I can use the primitive function **list**

```
(list 1 2 'serpico)
```

```
'(1 2 serpico)
```

Oh, and actually I can use this to represent trees too


1
2
3
4

How would I build this?

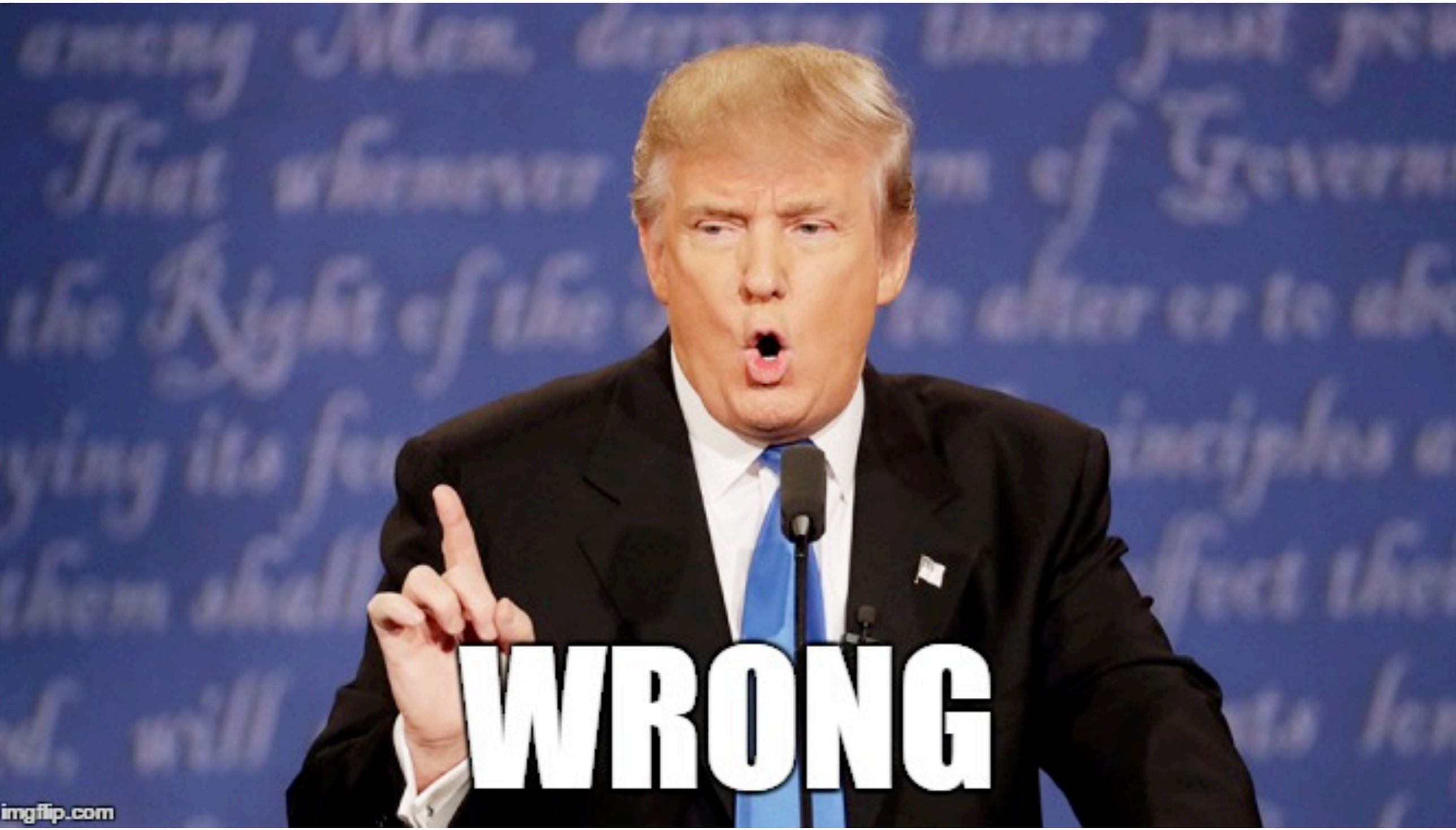
```
(define empty-tree 'empty-tree)

(define (make-leaf num) num)
```

```
(define (make-tree left right)
          (cons left right))
```

You define `(left-subtree tree)`

```
(define (least-element tree)
 (if (number? tree)
     tree
     (least-element (left-subtree tree))))
```

But surely I need things like numbers right?


WRONG
imgflip.com

It turns out, you could build those using just

cons, car, cdr, if, =, and ‘()

Define the number n as …

‘()

‘(())

‘(() ())

…

```
(define (weird-plus i j)
  (if (equal? i '())
      j
      (weird-plus (cdr i)
                  (cons '() j))))
```

```
(weird-plus '(() ()) '(() ()))

'(() () () ())
```

It turns out, if I'm clever, we can even get rid of **if** and **equal**

(Though we shall not do so here..)

I can build my own datatypes in this manner

I usually write **constructor** functions to help me build datatypes

I usually write **constructor** functions to help me build datatypes

And I usually write **destructor** functions to access it

```
(define (make-complex real imag)
  (cons real imag))
```

And I usually write **destructor** functions to access it

```
(define (make-complex real imag)
  (cons real imag))

(define (get-real complex)
  (car complex))

(define (get-imag complex)
  (cdr complex))
```

Now, define `(add-complex c1 c2)`

Next, define `(make-cartesian x y)`

And the associated helper functions

Next class we will talk about…

struct

match

I/O

And switch over to layout in assembly