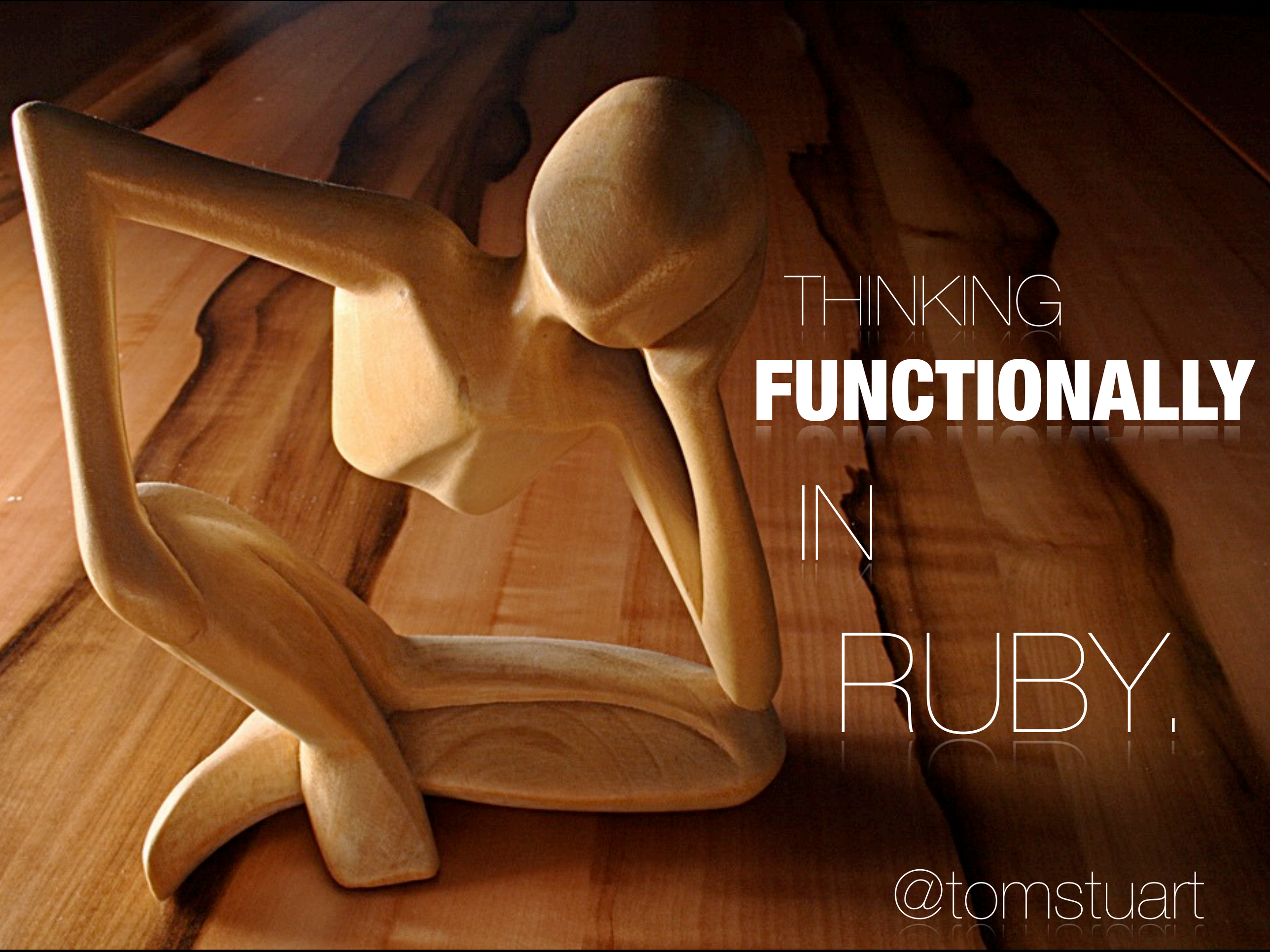


THINKING FUNCTIONALLY IN RUBY.

@tomstuart

1: Functional programming is a pretty neat idea.

2: `Enumerable` contains some useful methods.

I hope
these things
are connected.

1.

(Functional programming is a pretty neat idea)

What is functional programming?

A programming style.

Two broad families of languages:

- Scheme
- Common Lisp
- Dylan
- Clojure

1. Lisp-like

- Dynamically typed
- Homoiconic (code is data)
- Lists are fundamental

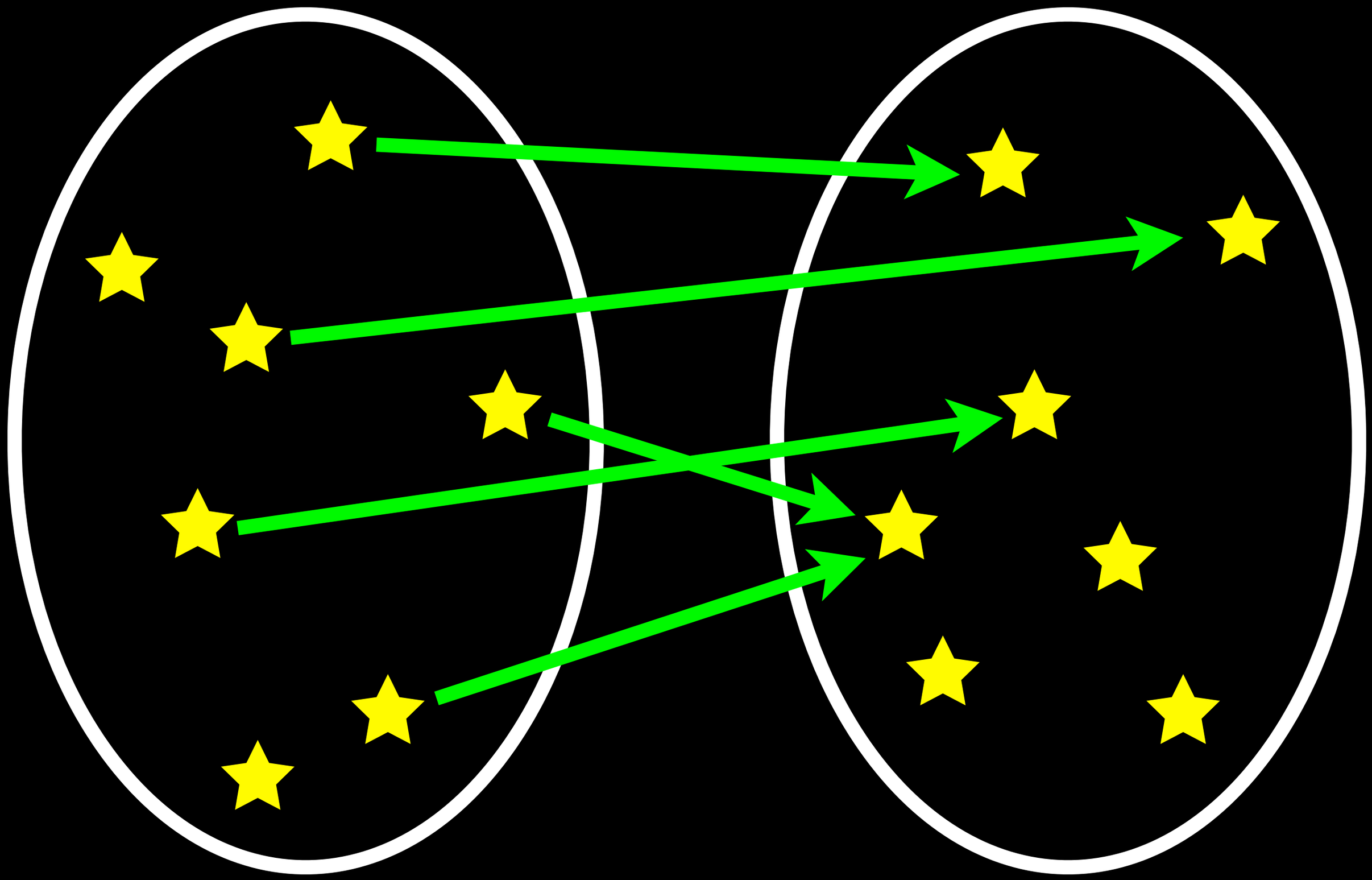
- Standard ML
- Haskell
- OCaml
- Scala (sort of)

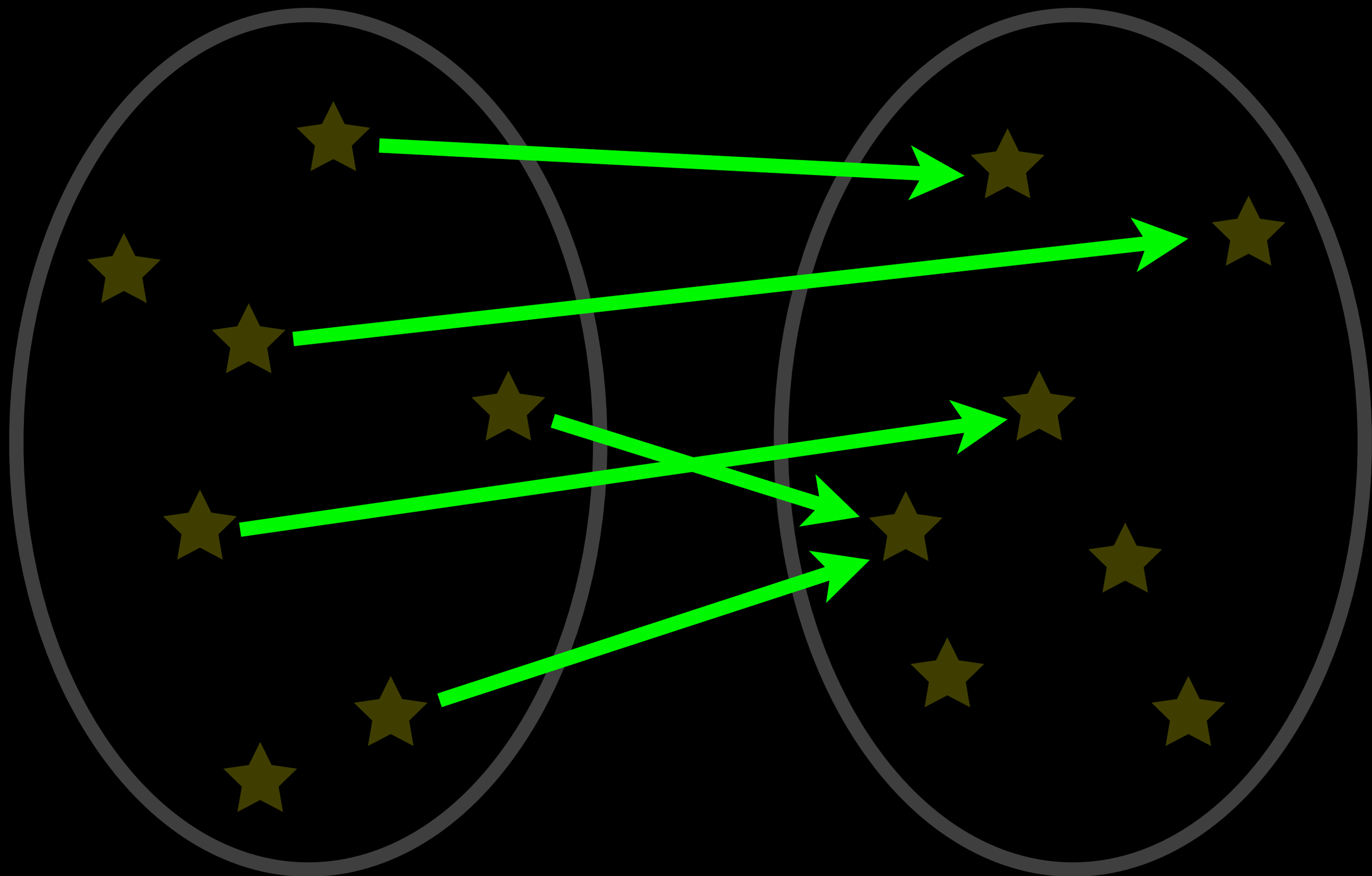
2. ML-like

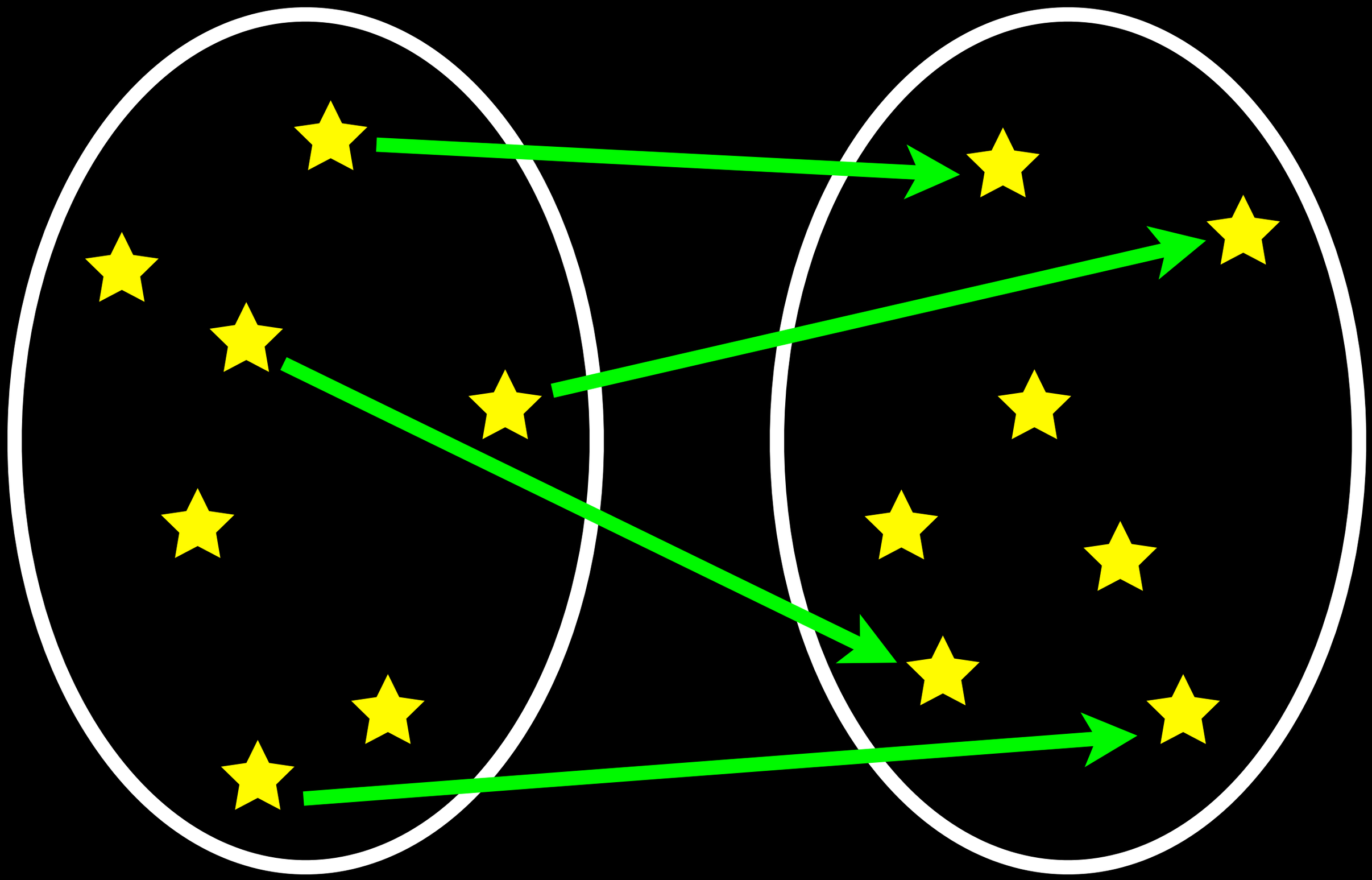
- Statically typed (+ reconstruction)
- Code is not data
- ADTs and pattern matching

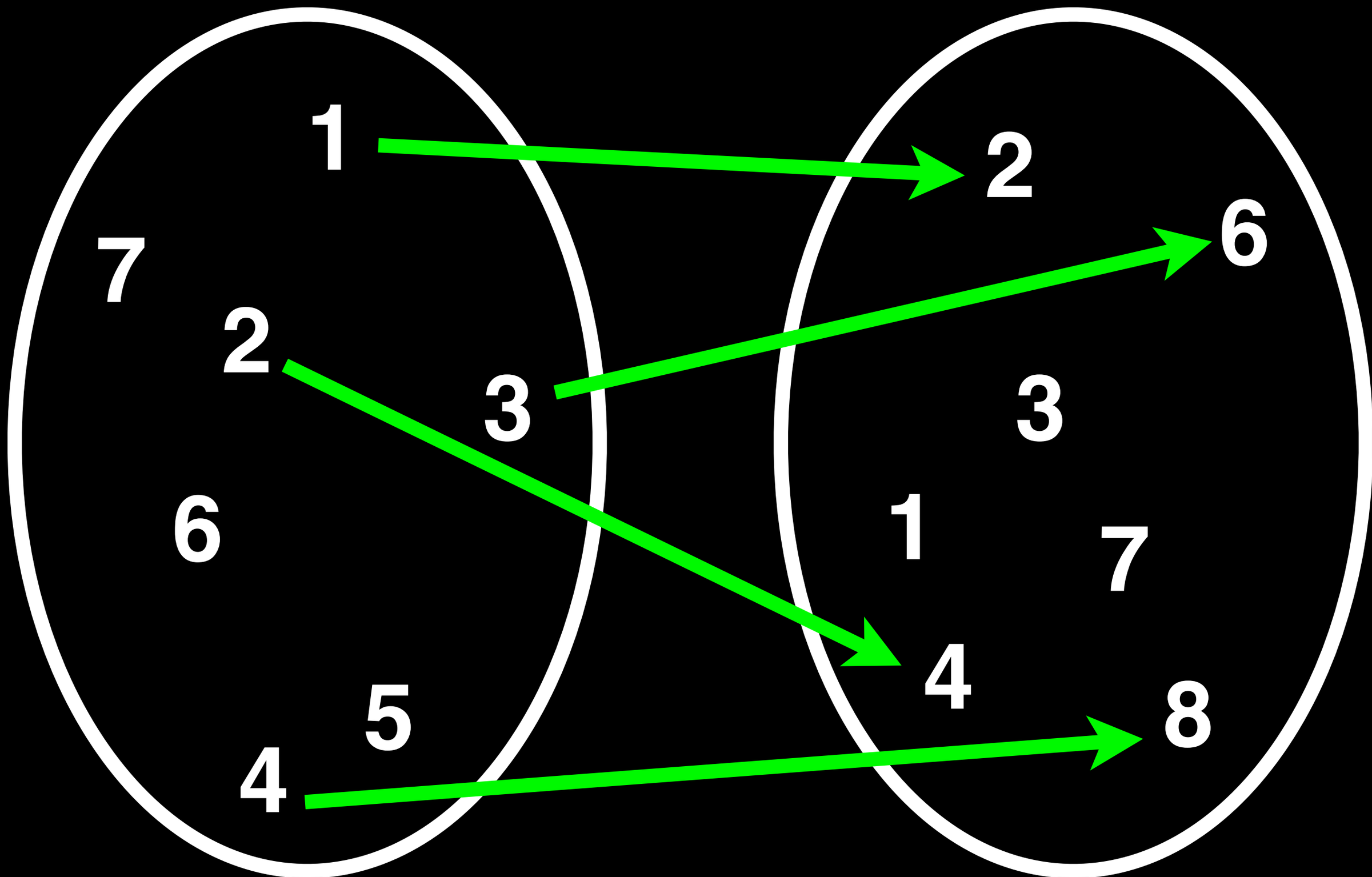
Functions as values

What is a function?








1
7
2
3
6
5
4
2
6
3
1
7
4
8

Functions as values

“First-class functions”

“closures”, “lambdas”,
“anonymous functions”

Implies: higher-order functions

No side effects

Values are immutable

All functions do is return their result

No state

No I/O

(In reality,
different languages
accomplish this
to varying degrees.)

Implies: recursion

Implies: persistent data structures

So what?

Unfortunately...

Declarative programming

WHAT!

is counterintuitive when you're accustomed to

imperative programming

HOW!

Copying is expensive

But!

Referential transparency

"an expression can be replaced with its value"

- Good for efficiency
  - caching/memoisation/inlining
  - ultimately more efficient
- Good for programmer understanding
  - state is incredibly hard to deal with
  - ultimately more intuitive
- Good for code reuse and testing

Highly expressive

Strongly compositional

Deeply satisfying

Future-proof

- Moore's Law is running out of steam
  - similar transistor density, more cores
- The future's concurrent
- Concurrent access to mutable state is hard...
  - but not if you don't have mutable state!
- Parallelising an algorithm is hard...
  - but it's easier if your code isn't overspecified

OK seriously:

so what?

Ruby can do some of this.

Not really a functional language, but we can pretend and get some of the benefits.

Use function values

blocks, procs

```
lambda { |x| x + 1 }
```

Consider treating your values as immutable

State is rarely worth it.

It will kill you in the end.

http://clojure.org/state

Use the functional-flavoured parts of the standard library

Think functionally.

Be declarative.

What, not how.

2.

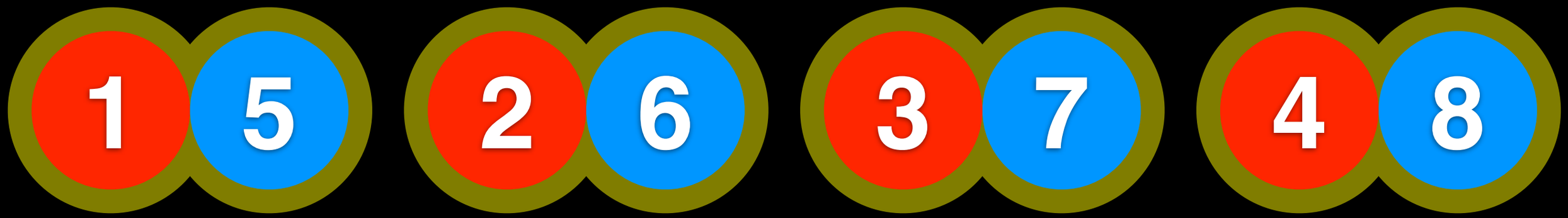
(`Enumerable` contains some useful methods)

Enumerable#zip

```
[1, 2, 3, 4].
zip([5, 6, 7, 8])
```

1 2 3 4

5 6 7 8


1
5
2
6
3
7
4
8

$$[[1,5],[2,6],[3,7],[4,8]]$$

```
[1, 2, 3, 4].
zip([5, 6, 7, 8],
    [9, 10, 11, 12])
```

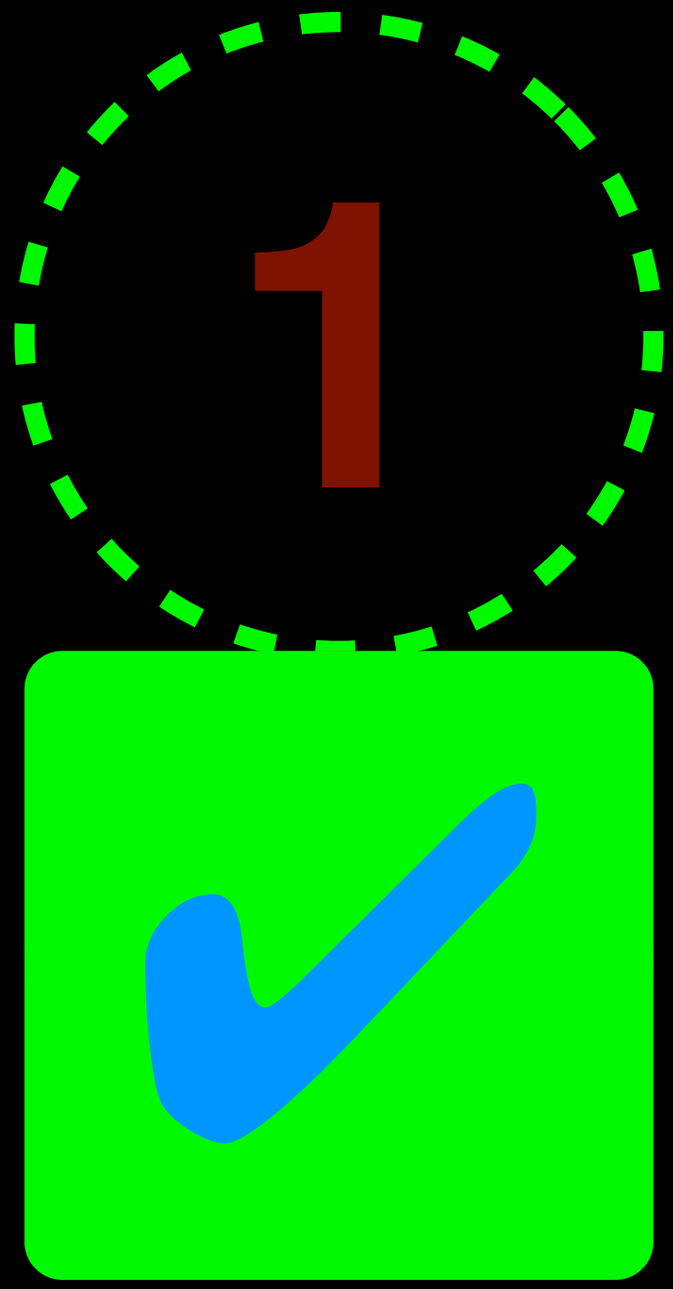
`Enumerable#select`

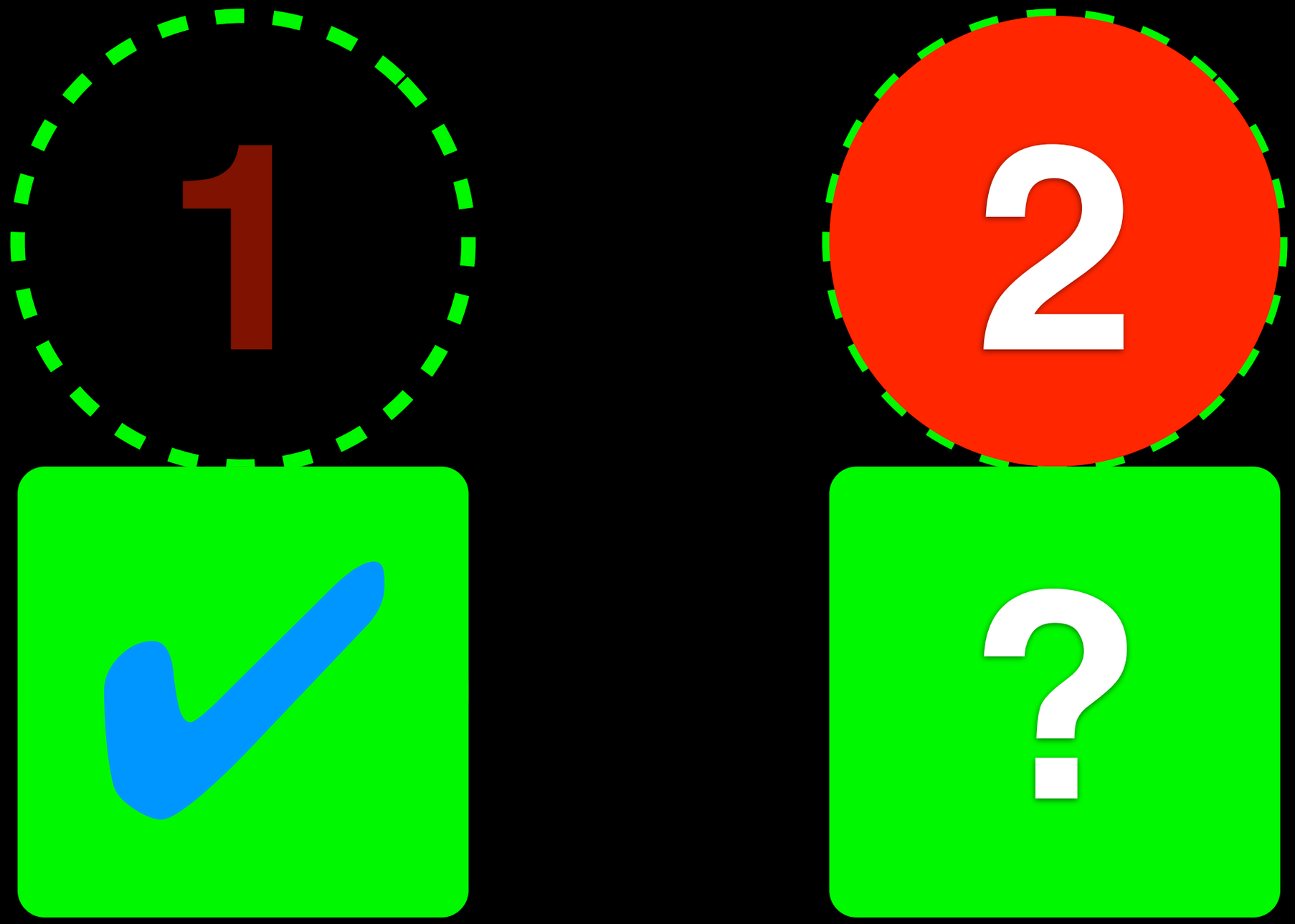
(a.k.a. `#find_all`)

```
{ |x| x.odd? }
```

1

?

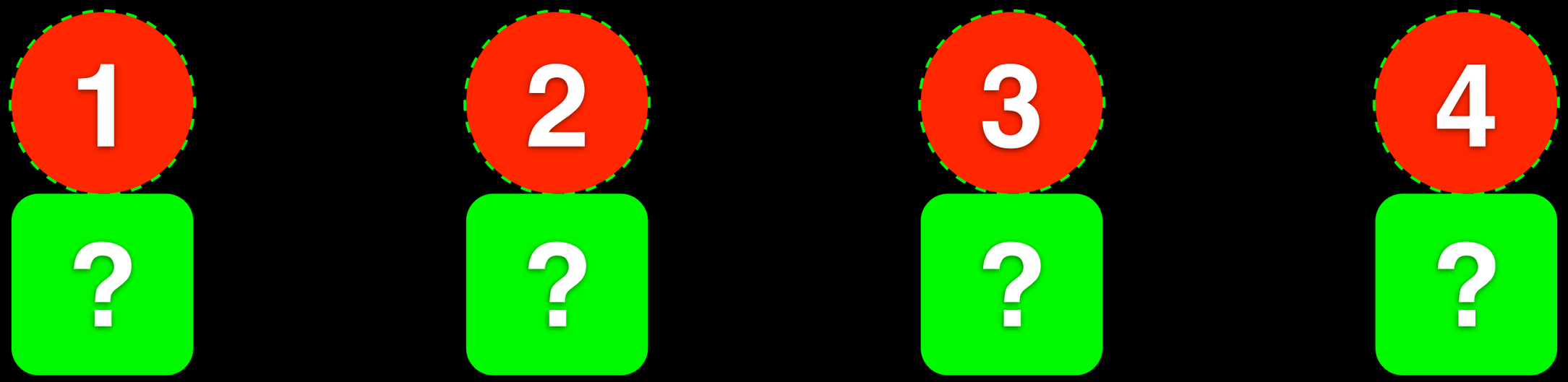

1


1
2
?

1

2

```
[1, 2, 3, 4].
select { |x| x.odd? }
```

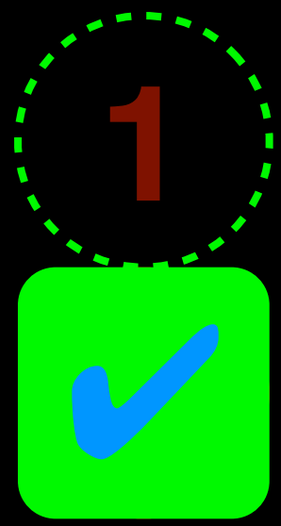

1
2
3
4
?
?
?
?

1 ✓

2 ✗

3 ✓

4 ✗


1

1

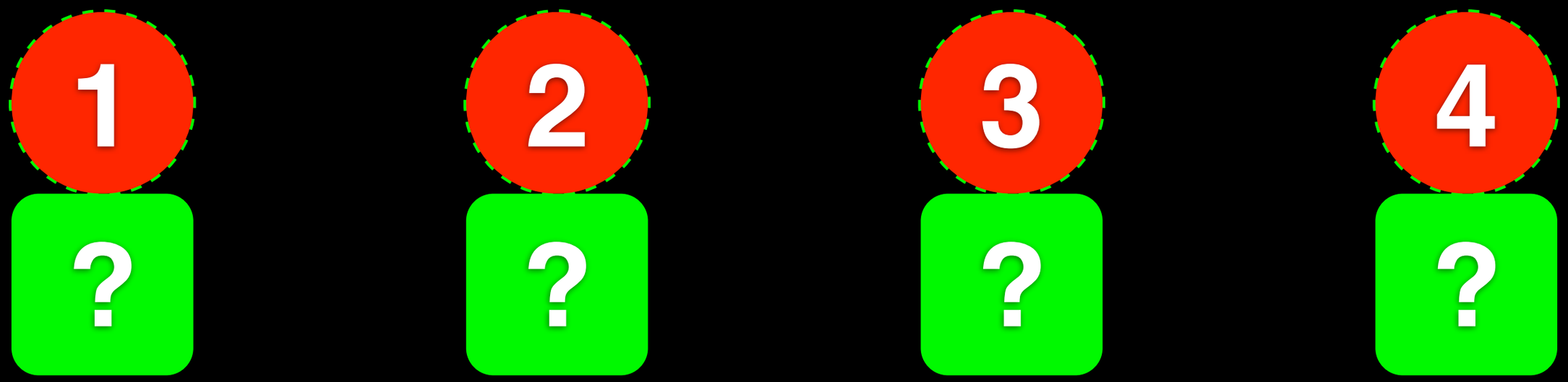
3

3

[1 , 3]

Enumerable#partition

```
[1, 2, 3, 4].
  partition { |x| x.odd? }
```

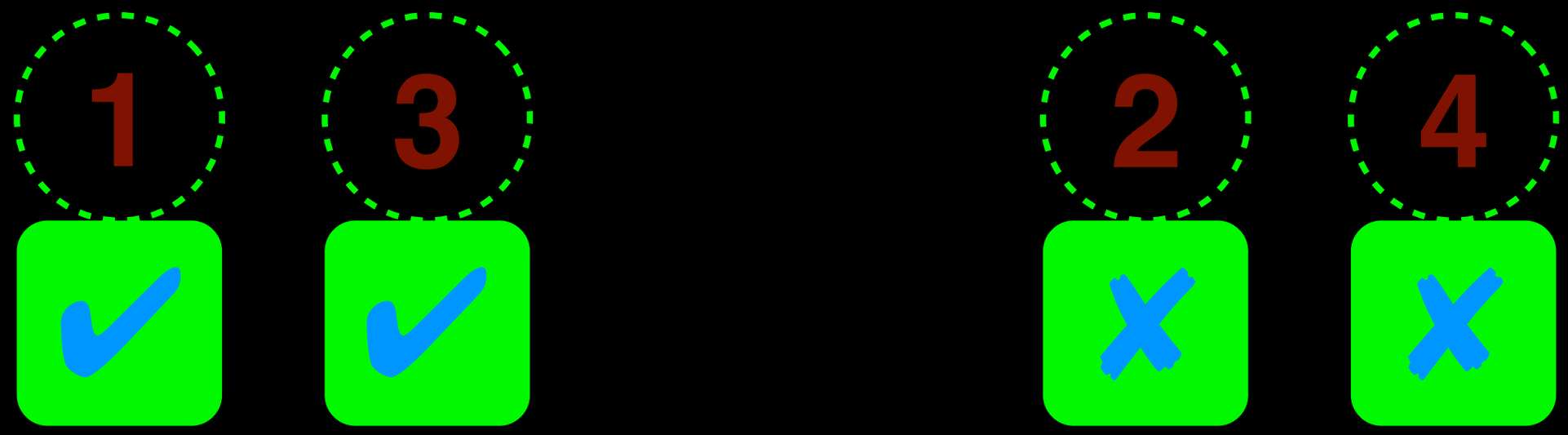

1
2
3
4
?
?
?
?

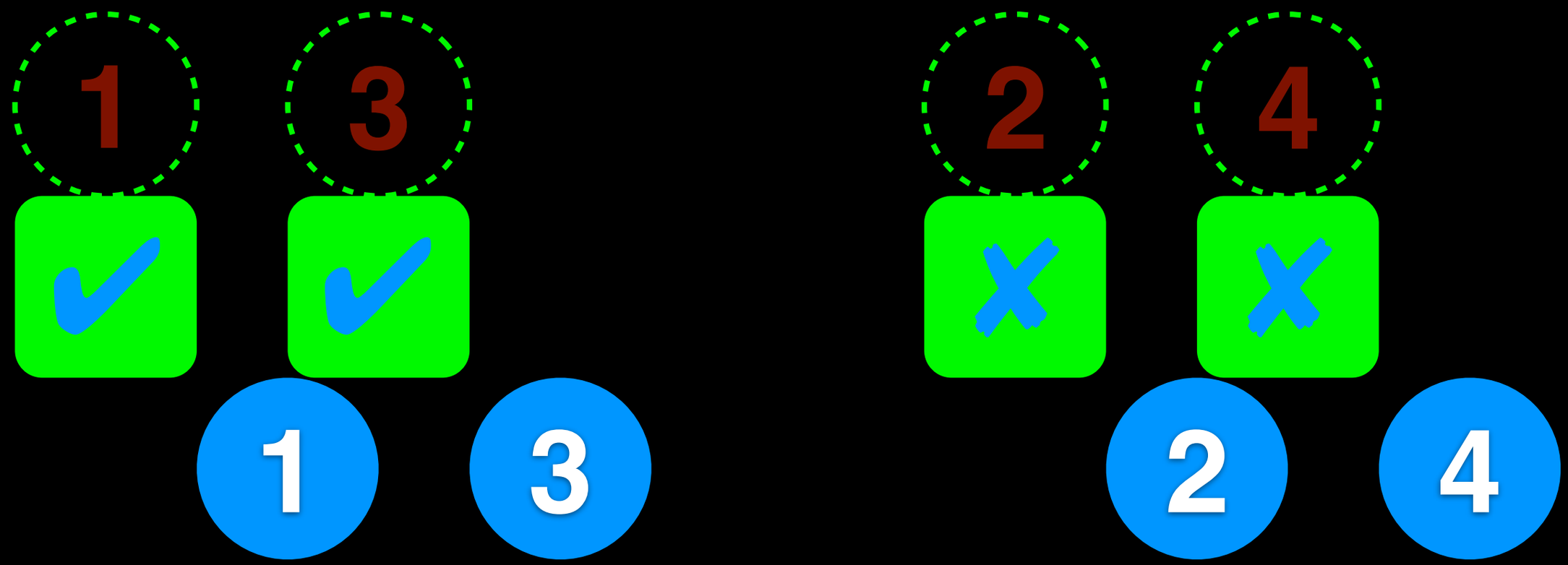
1 ✓

2 ✗

3 ✓

4 ✗


1
3
2
4


1
3
2
4
1
3
2
4

[[1, 3], [2, 4]]

Enumerable#map

(a.k.a. #collect)

```
{ |x| x * 3 }
```

2

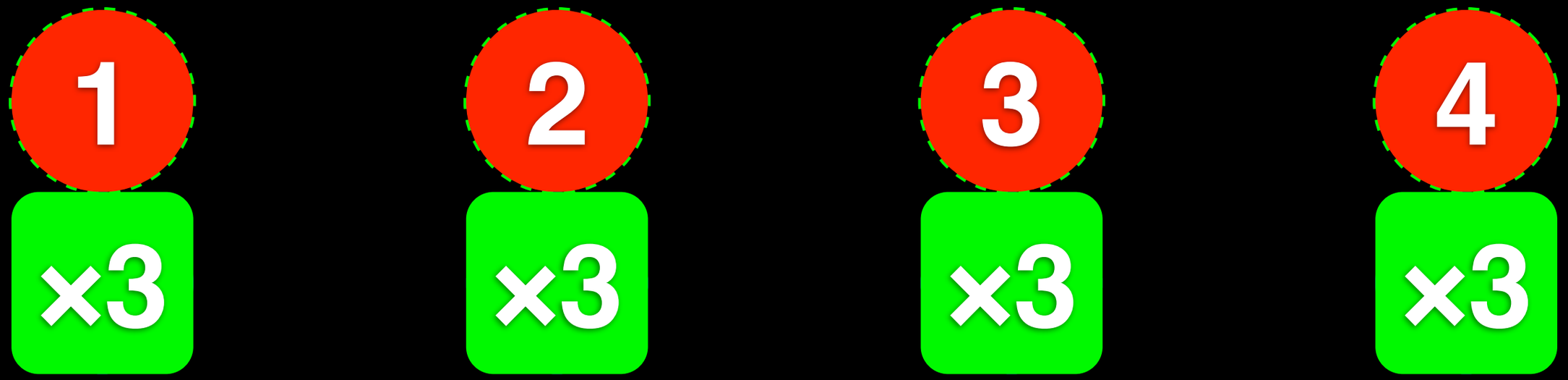
×3

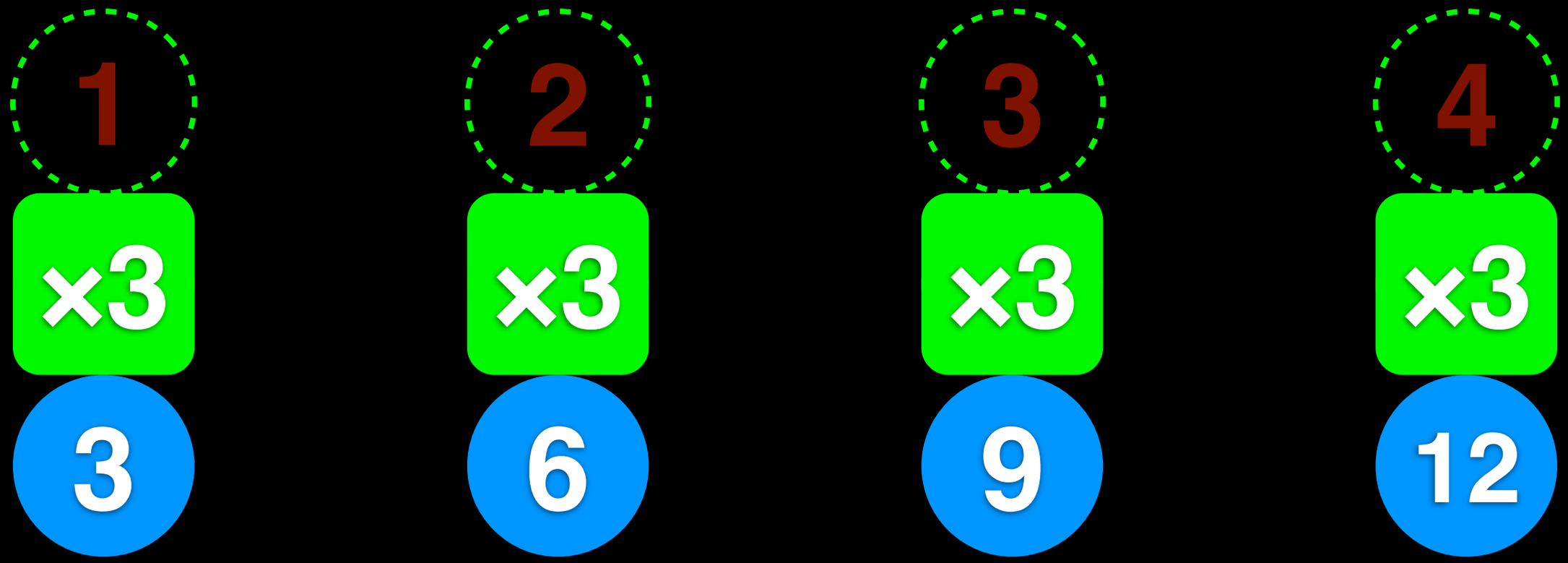
2

×3

6

```
[1, 2, 3, 4].
map { |x| x * 3 }
```


1
×3
2
×3
3
×3
4
×3


1
×3
3
2
×3
6
3
×3
9
4
×3
12

[3 6 9 12]

Enumerable#inject

(a.k.a. #reduce)

```
{ |x, y| x + y }
```

3 5

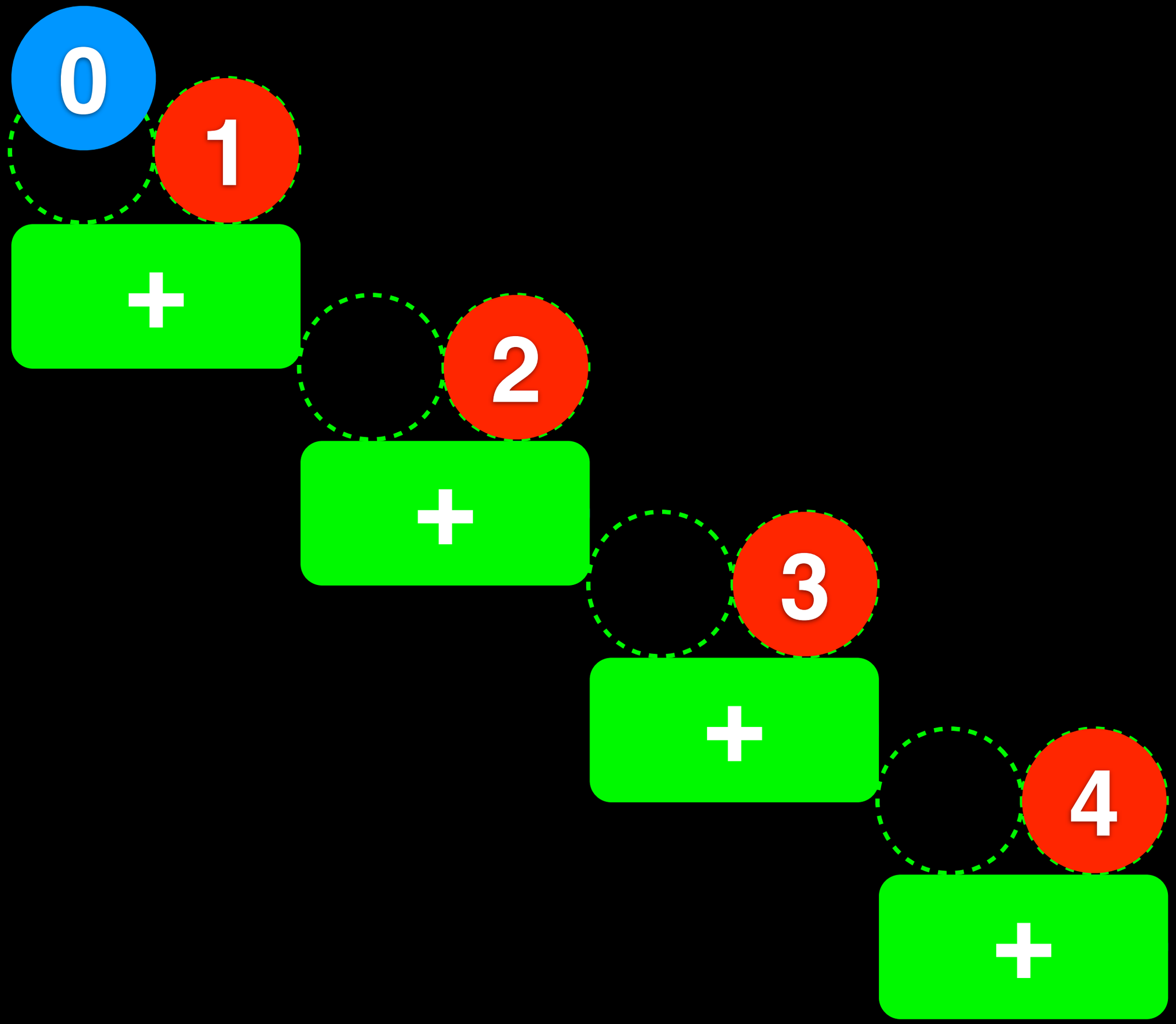
+

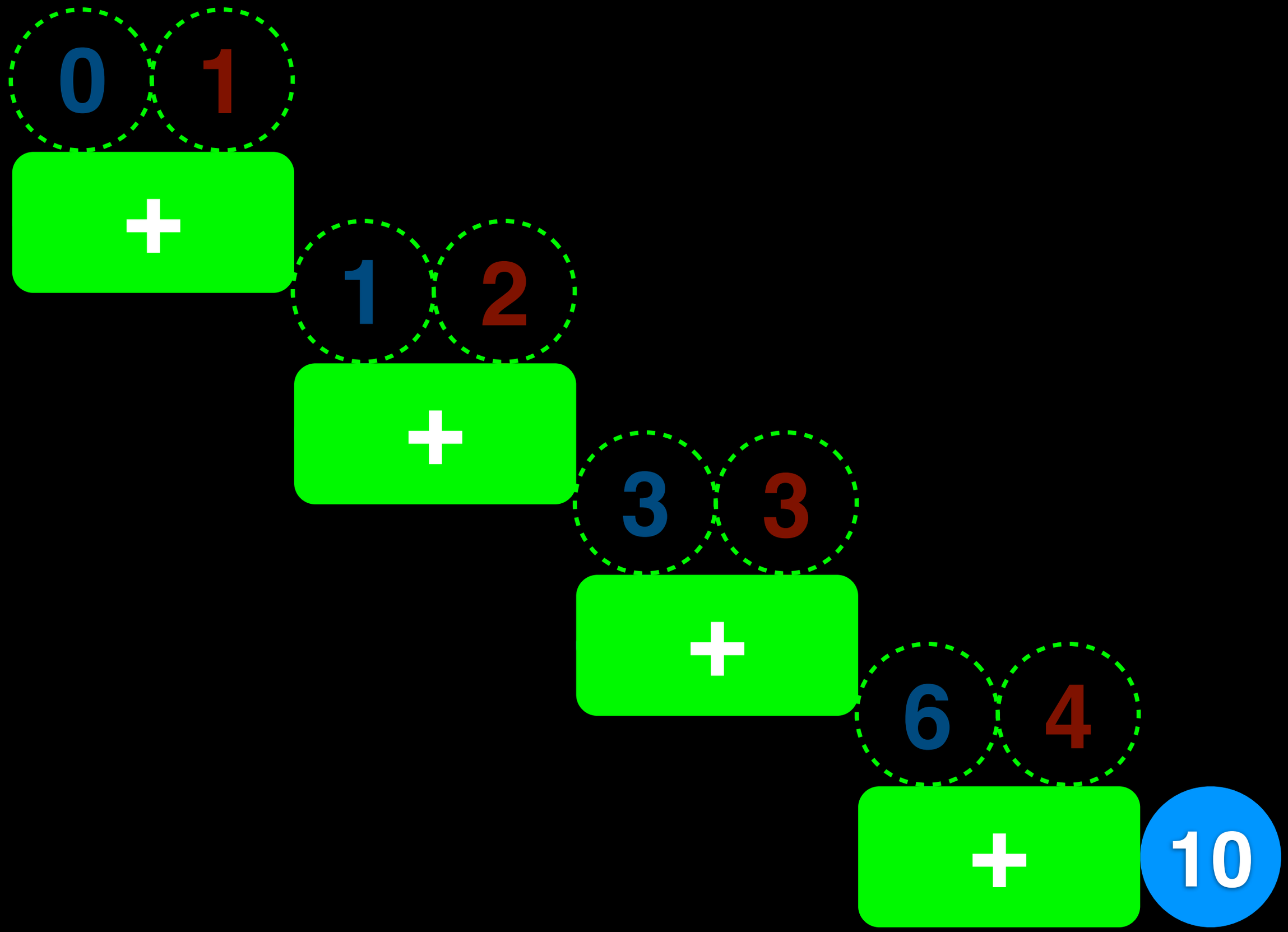
3 5

+

8

```
[1, 2, 3, 4].
  inject(0) { |x,y| x+y }
```


0
1
+
2
+
3
+
4
+


0
1
+
1
2
+
3
3
+
6
4
+
10

```ruby
module Enumerable
  def inject(initial)
    result = initial

    for element in self
      result = yield(result, element)
    end

    result
  end
end
```

a.k.a. “left fold”

(`foldl`, `fold_left`)

0	1	2	3	4

10

```
(((0 + 1) + 2) + 3) + 4
```

...versus “right fold”

(`foldr`, `fold_right`)

1	2	3	4	0

10

```
1 + (2 + (3 + (4 + 0)))
```

The initial argument is optional...

```
[1, 2, 3, 4].
inject { |x,y| x+y }
```

```
[2, 3, 4].
  inject(1) { |x,y| x+y }
```

...but **only** if the output is the same type as the input...

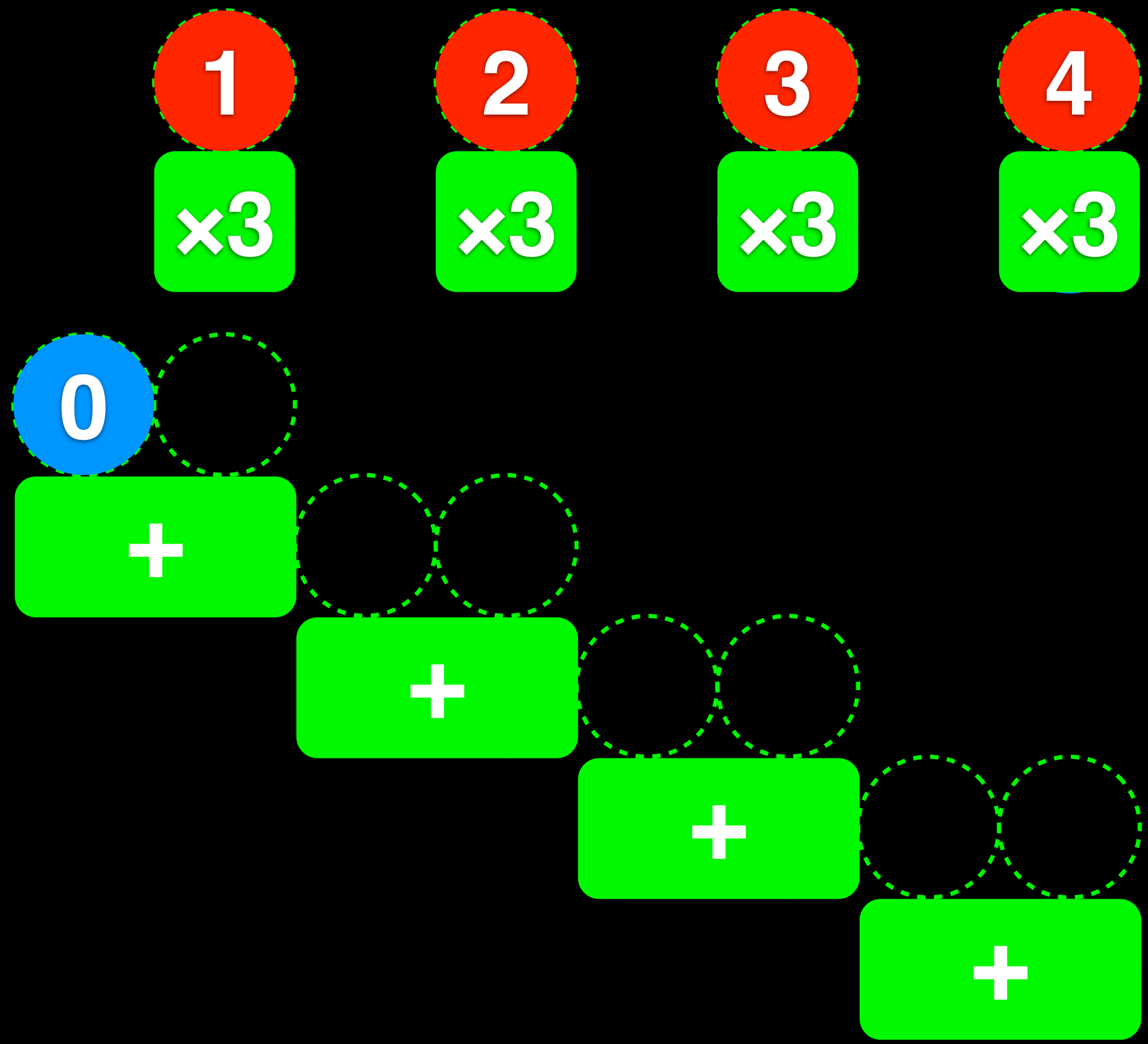
```
>> ['El', 'rug'].
   inject(0) { |l,s| l + s.length }
=> 5


>> ['El', 'rug'].
   inject { |l,s| l + s.length }
TypeError: can't convert Fixnum into String
  from (irb):1:in `+'
  from (irb):1
  from (irb):1:in `inject'
  from (irb):1:in `each'
  from (irb):1:in `inject'
  from (irb):1
```

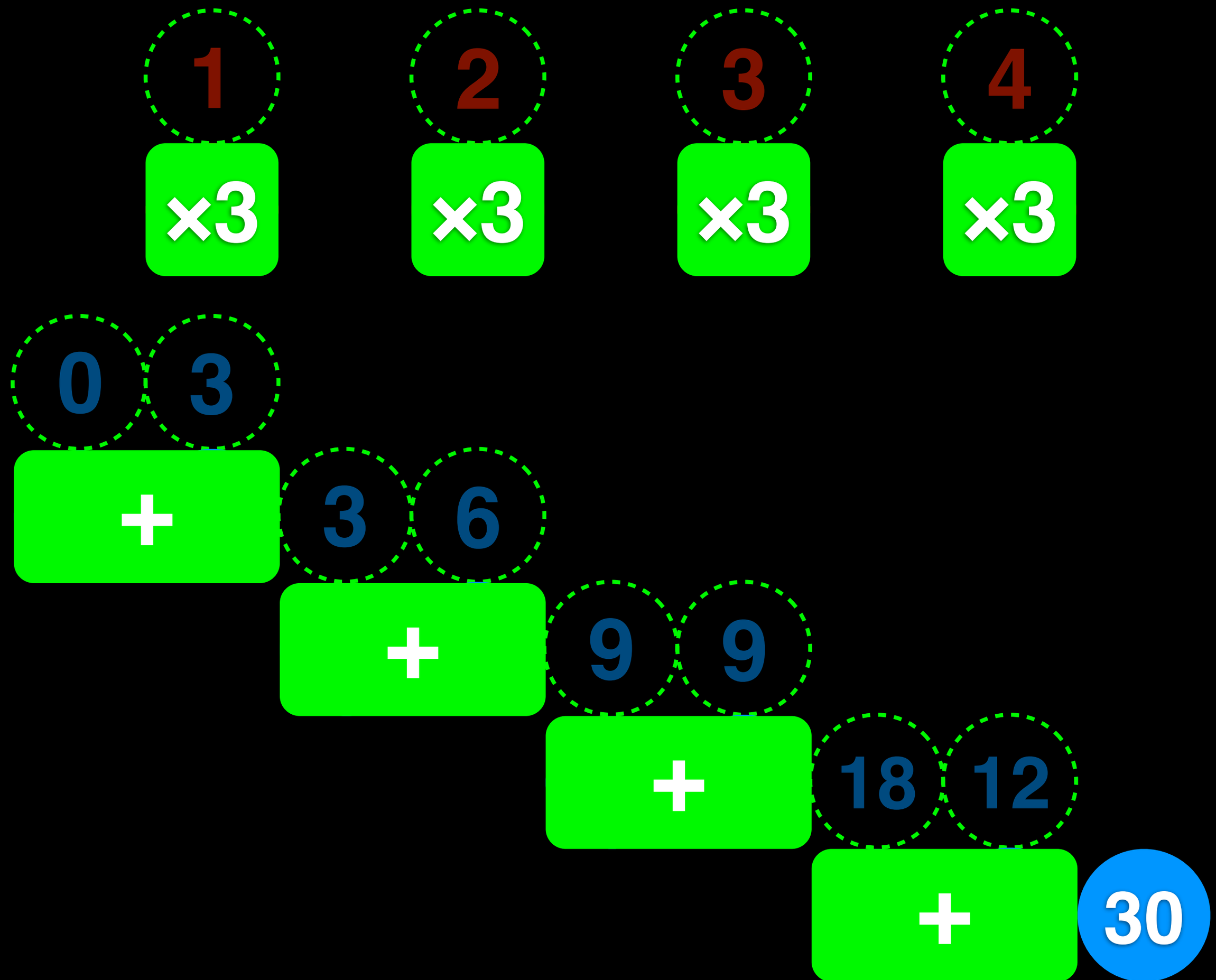
…**and** it’s meaningful to get `nil` when the collection is empty

```
>> [].inject { |x,y| x+y }
=> nil

>> [].inject(0) { |x,y| x+y }
=> 0

>> [].inject(1) { |x,y| x*y }
=> 1
```

COMPOSE!

```
[1, 2, 3, 4].
  map { |x| x * 3 }.
  inject(0) { |x| x+y }
```


1
×3
2
×3
3
×3
4
×3
0
+
+
+
+


1
2
3
4
×3
×3
×3
×3
0
3
+
3
6
+
9
9
+
18
12
+
30

I am so excited.

What now?

Review your Ruby code.

Func it up.

```ruby
result = ''
for name in names
  unless result.empty?
    result << ', '
  end
  result << name
end
result
```

```
result = ''
for name in names
  unless result.empty?
    result << ', '
  end
  result << name
end
result
```

```
names.join(', ')
```

```ruby
def count_mines_near(x, y)
  count = 0
  for i in x-1..x+1
    for j in y-1..y+1
      count += 1 if mine_at?(i, j)
    end
  end
  count
end
```

```
def count_mines_near(x, y)
  count =
  for i in x-1..x+1
    for j y-1..y+1
      count += 1 if mine_at?(i, j)
    end
  end
  count
end

def count_mines_near(x, y)
  ((x-1..x+1).entries * 3).sort.
    zip((y-1..y+1).entries * 3).
    select { |x, y| mine_at?(x, y) }.
    length
end
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3].sort =
[1, 1, 1, 2, 2, 2, 3, 3, 3]
```

```
def count_mines_near(x, y) # = (2, 8)
  ((x-1..x+1).entries * 3).sort.
    zip((y-1..y+1).entries * 3).
    select { |x, y| mine_at?(x, y) }.
    length
end
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3].sort =
[1, 1, 1, 2, 2, 2, 3, 3, 3].zip(
[7, 8, 9, 7, 8, 9, 7, 8, 9]) =
[[1, 7], [1, 8], [1, 9],
 [2, 7], [2, 8], [2, 9],
 [3, 7], [3, 8], [3, 9]]
```

```
def count_mines_near(x, y) # = (2, 8)
  ((x-1..x+1).entries * 3).sort.
    zip((y-1..y+1).entries * 3).
    select { |x, y| mine_at?(x, y) }.
    length
end
```

```
[1, 2, 3, 1, 2, 3, 1, 2, 3].sort =

[1, 1, 1, 2, 2, 2, 3, 3, 3].zip(

[7, 8, 9, 7, 8, 9, 7, 8, 9]) =

[[1, 7], [1, 8], [1, 9],
 [2, 7], [2, 8], [2, 9],
 [3, 7], [3, 8], [3, 9]].select {…} =

  [[1, 8], [3, 7]].length = 2

def count_mines_near(x, y) # = (2, 8)
  ((x-1..x+1).entries * 3).sort.
    zip((y-1..y+1).entries * 3).
    select { |x, y| mine_at?(x, y) }.
    length
end
```

```
[   [1, 7],    [1, 8], …,    [1, 1007],
    [2, 7],    [2, 8], …,    [2, 1007],
            …,                …,                 …,
 [1000, 7], [1000, 8], …, [1000, 1007],
 [1001, 7], [1000, 8], …, [1001, 1007]]
```

```
def count_mines_near(x, y)
  ((x-500..x+500).entries * 1001).sort.
    zip((y-500..y+500).entries * 1001).
    select { |x, y| mine_at?(x, y) }.
    length
end
```

[[1, 7]
[2, 7]
…
[1, 8], …
[2, 8], …
…
[1, 1007]
[2, 1007]
…
[1000, 7], [1000, 8],
[1001, 7], [1000, 8],
…, [1000, 1007]
…, [1001, 1007]
173
96
121
78
237
705

```ruby
def numbers_near(n, radius)
  ((n - radius)..(n + radius)).entries
end

def squares_near(x, y, radius)
  diameter = (radius * 2) + 1
  (numbers_near(x, radius) * diameter).
    sort.
    zip(numbers_near(y, radius) * diameter)
end

def count_mines_near(x, y, radius = 1)
  squares_near(x, y, radius).
    select { |x, y| mine_at?(x, y) }.
    length
end
```

Learn a functional language

- OCaml
- Scheme (Google “SICP”)
- Clojure
- Scala
- Haskell? Erlang?

Thanks!

@tomstuart / tom@experthuman.com

http://www.flickr.com/photos/somemixedstuff/2403249501/

http://en.wikipedia.org/wiki/File:Modified-pc-case.png