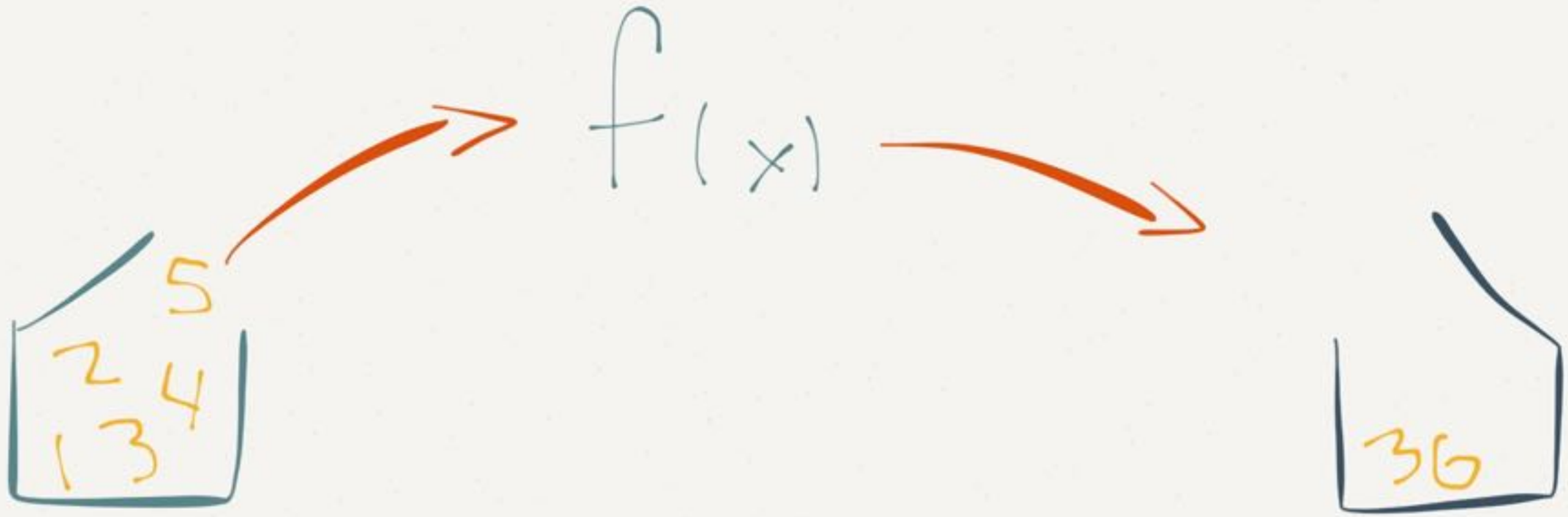
Functors, Monads and Other Made-Up Words

Functors

Mappable

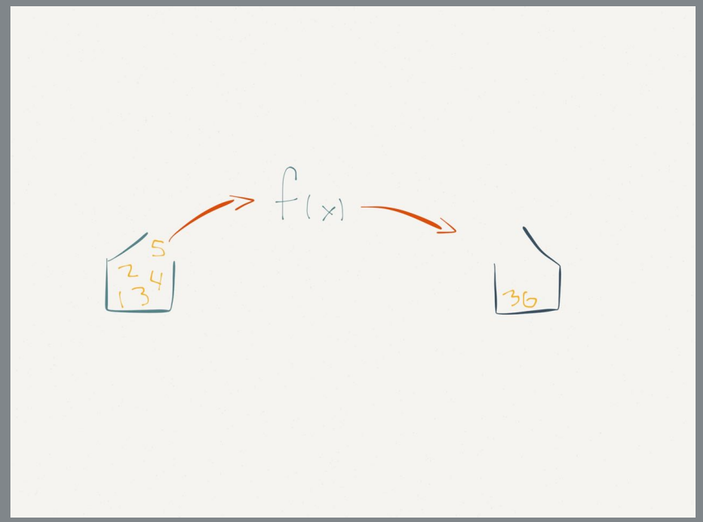
```swift
let numbers = [1, 2, 3, 4, 5]
var squares: [Int] = []

for number in numbers {
    squares.append(number * number)
}

// [1, 4, 9, 16, 25]
```

```
let mappedSquares = map(numbers, { number in
    return number * number
})

// [1, 4, 9, 16, 25]
```

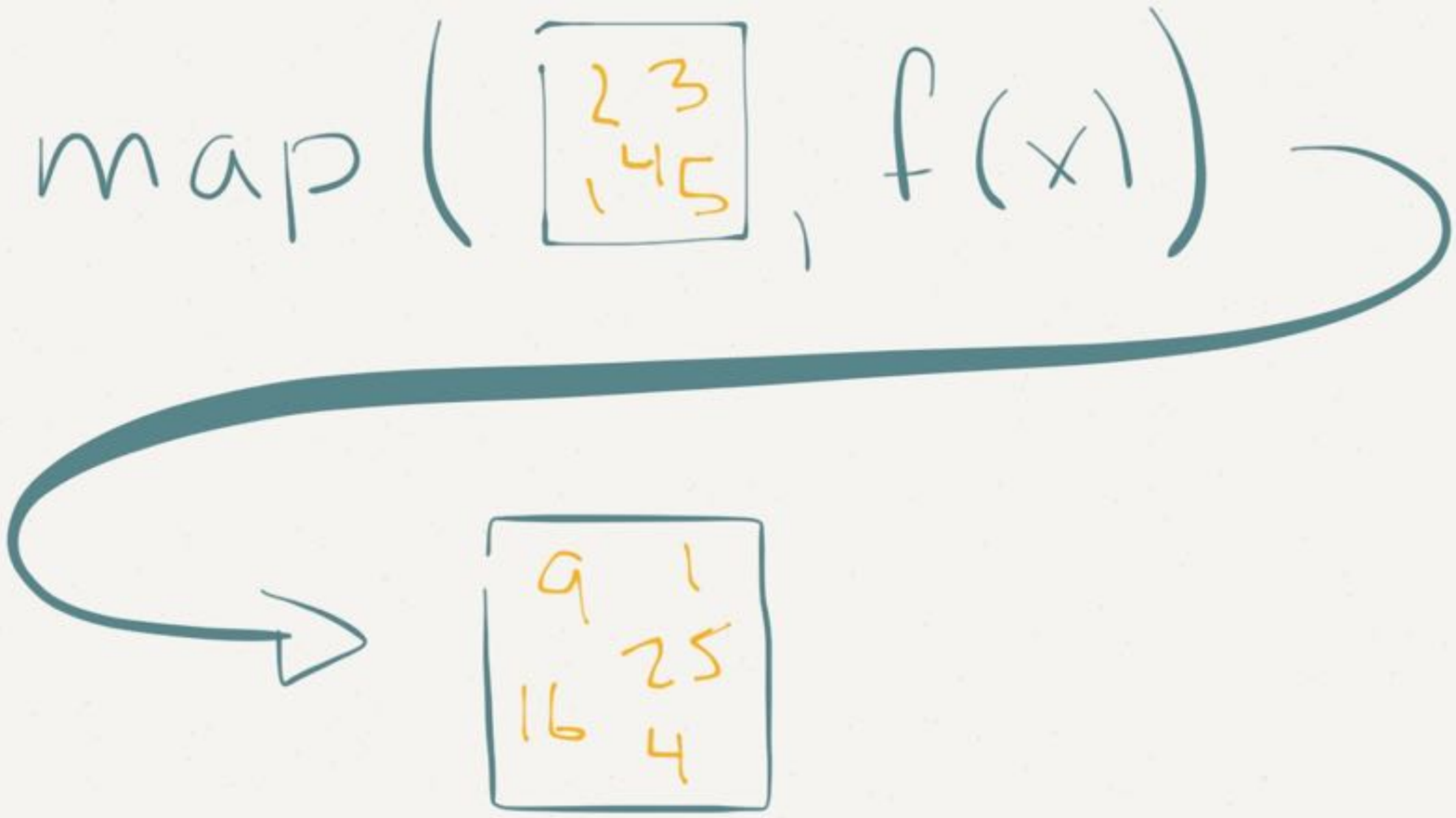

f(x)
5
2 4
1 3
36

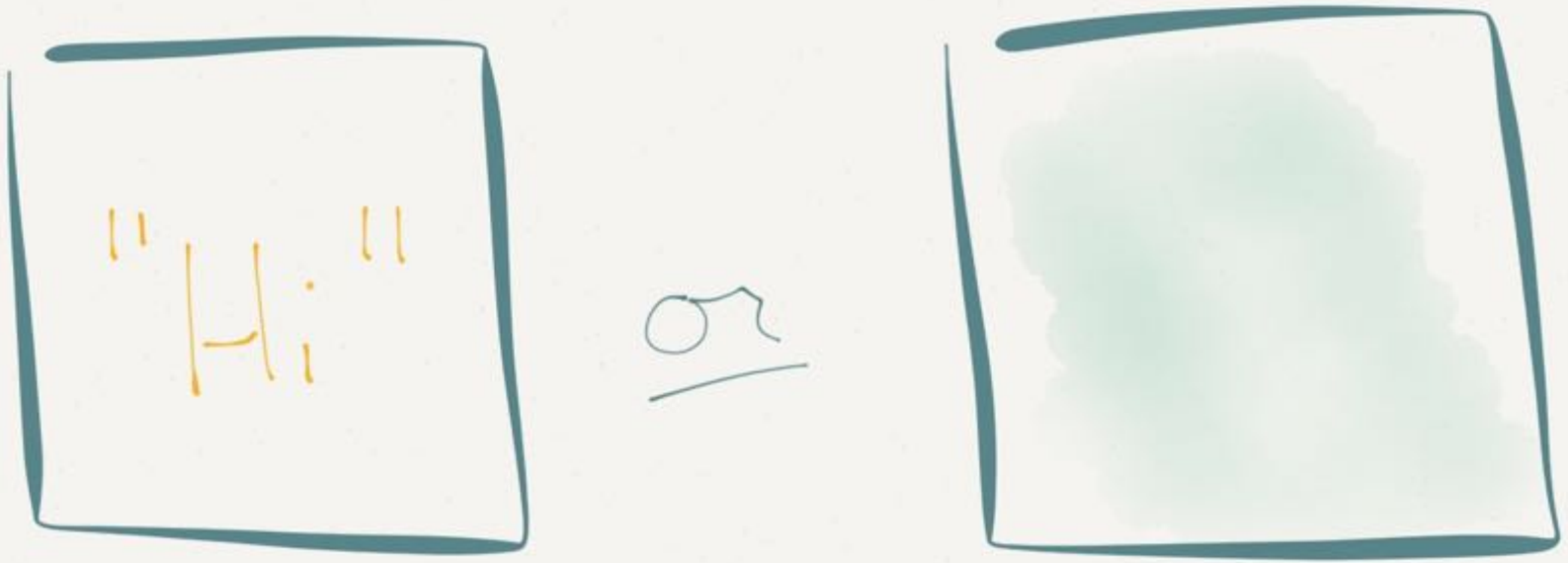
```
func map<I, O>(array: [I], transform: (I) -> O) -> [O] {
    var output: [O] = []
    for input in array {
        output.append(transform(input))
    }
    return output
}
```

```
func map<I, O>(array: [I], transform: (I) -> O) -> [O] {
    var output: [O] = []
    for input in array {
        output.append(transform(input))
    }
    return output
}
```

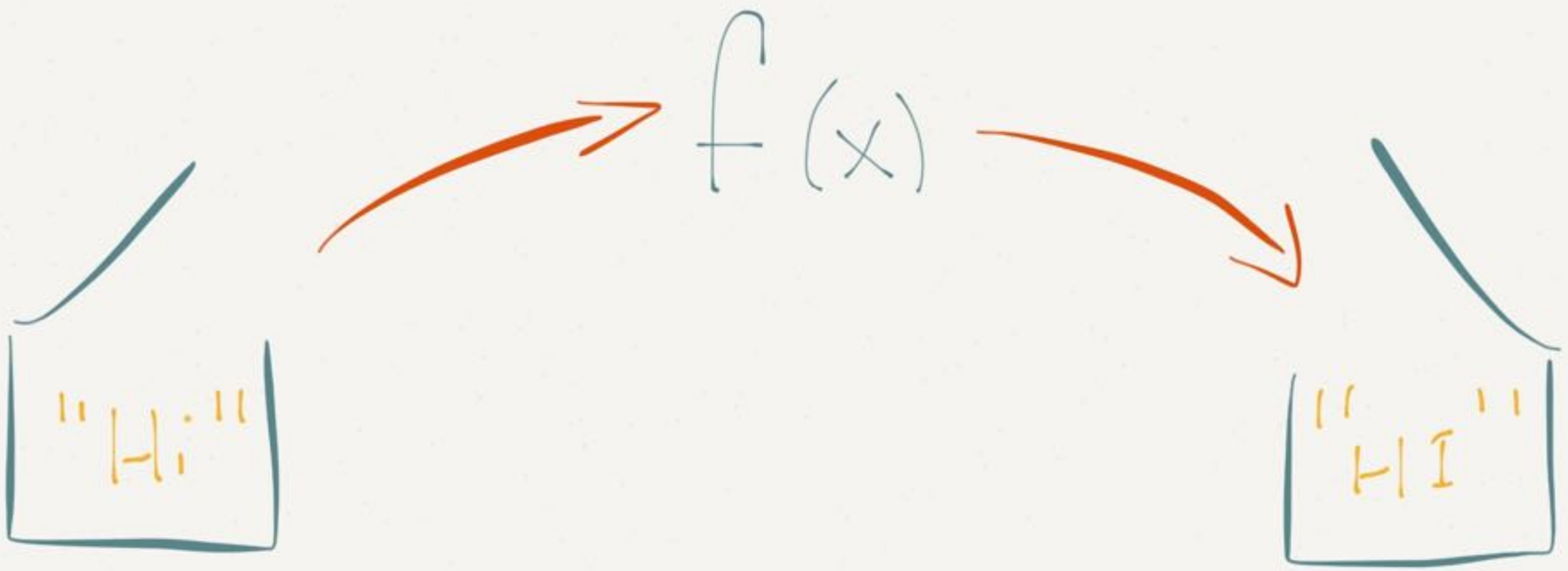


```
map(numbers, { number in number * number })

// or:

numbers.map({ number in number * number })
```


map(2 3 1 4 5 , f(x))
9 1 25 16 4


"Hi"
or

```
func square(num: Int) -> Int { return num * num }

func calculatePasscode(input: String?) -> Int? {
    let passcode: Int?
    if let i = input {
        let number = count(i)
        passcode = square(number)
    }
    else {
        passcode = Optional.None
    }
    return passcode
}
```


f(x)
"Hi"
"HI"

$f(x)$

```
// Array
func map<I, O>(array: [I], transform: (I) -> O) -> [O]

// Optional
func map<I, O>(optional: I?, transform: (I) -> O) -> O?
```

```
func map<I, O>(optional: I?, transform: (I) -> O) -> O? {
    if let o = optional {
        return Optional.Some(transform(o))
    }
    return Optional.None
}
```

```
func calculatePasscode2(input: String?) -> Int? {
    let number = map(input, count)
    let passcode = map(number, square)
    return passcode
}
```

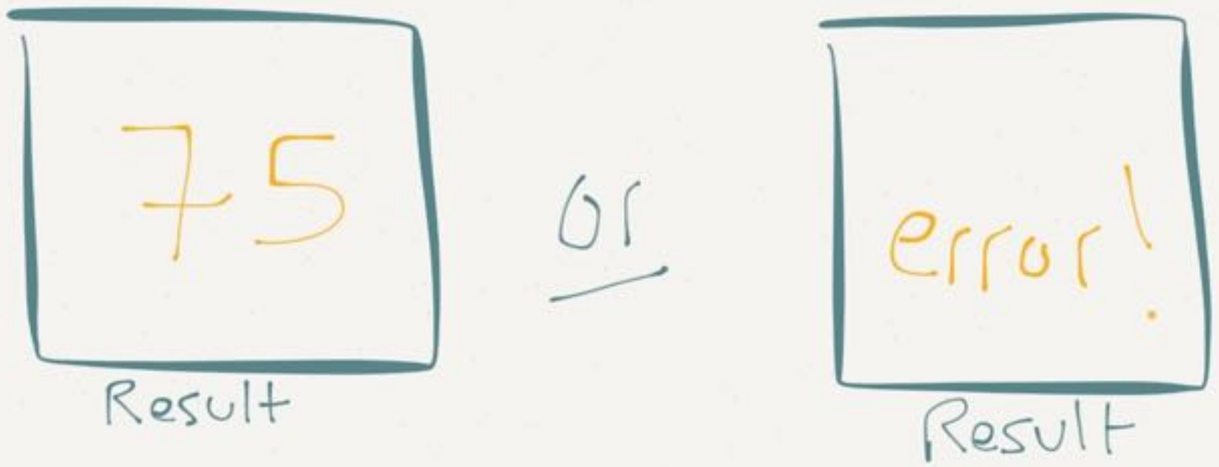
```
func calculatePasscode2(input: String?) -> Int? {
    return map(map(input, count), square)
}
```

```
func calculatePasscode2(input: String?) -> Int? {
    return map(map(input, count), square)
}

extension Optional {
    func map<I, O>(optional: I?, transform: (I) -> O) -> O? {
        if let o = optional {
            return .Some(transform(o))
        }
        return .None
    }
}
```

```
func calculatePasscode3(input: String?) -> Int? {
    return input.map(count).map(square)
}
```

```
let name: String? = "Steve"
let secret = calculatePasscode2(name)
// 25

let emptyName: String? = .None
let secret2 = calculatePasscode2(emptyName)
// nil
```

```
func map<T, U>(array:   [T], transform: (T) -> U) -> [U]
func map<T, U>(optional: T?, transform: (T) -> U) ->  U?
```


75
Result
or
error!
Result

```
class Box<T> {
    let unbox: T
    init(_ value: T) {
        self.unbox = value
    }
}

enum Result<T>: Printable {
    case Value(Box<T>)
    case Error(NSError)

    var description: String {
        switch self {
        case .Value(let box):
            return "\(box.unbox)"
        case .Error(let error):
            return error.description
        }
    }
}
```

```
func map<I, O>(result: Result<I>, transform: (I) -> O) -> Result<O> {
    switch result {
    case .Value(let v):
        return Result.Value(Box(transform(v.unbox)))
    case .Error(let e):
        return Result.Error(e)
    }
}
```

```
func map<T, U>(array: [T],          transform: (T) -> U) -> [U]
func map<T, U>(optional: T?,        transform: (T) -> U) -> U?
func map<T, U>(result: Result<T>, transform: (T) -> U) -> Result<U>
```

```
protocol Functor<T> {
    func map<U>(transform: T -> U) -> Self<U>
}

func map<F: Functor, T, U>(x: F<T>, transform: T -> U) -> F<U> {
    return x.map(transform)
}
```

Applicatives

Mappable'

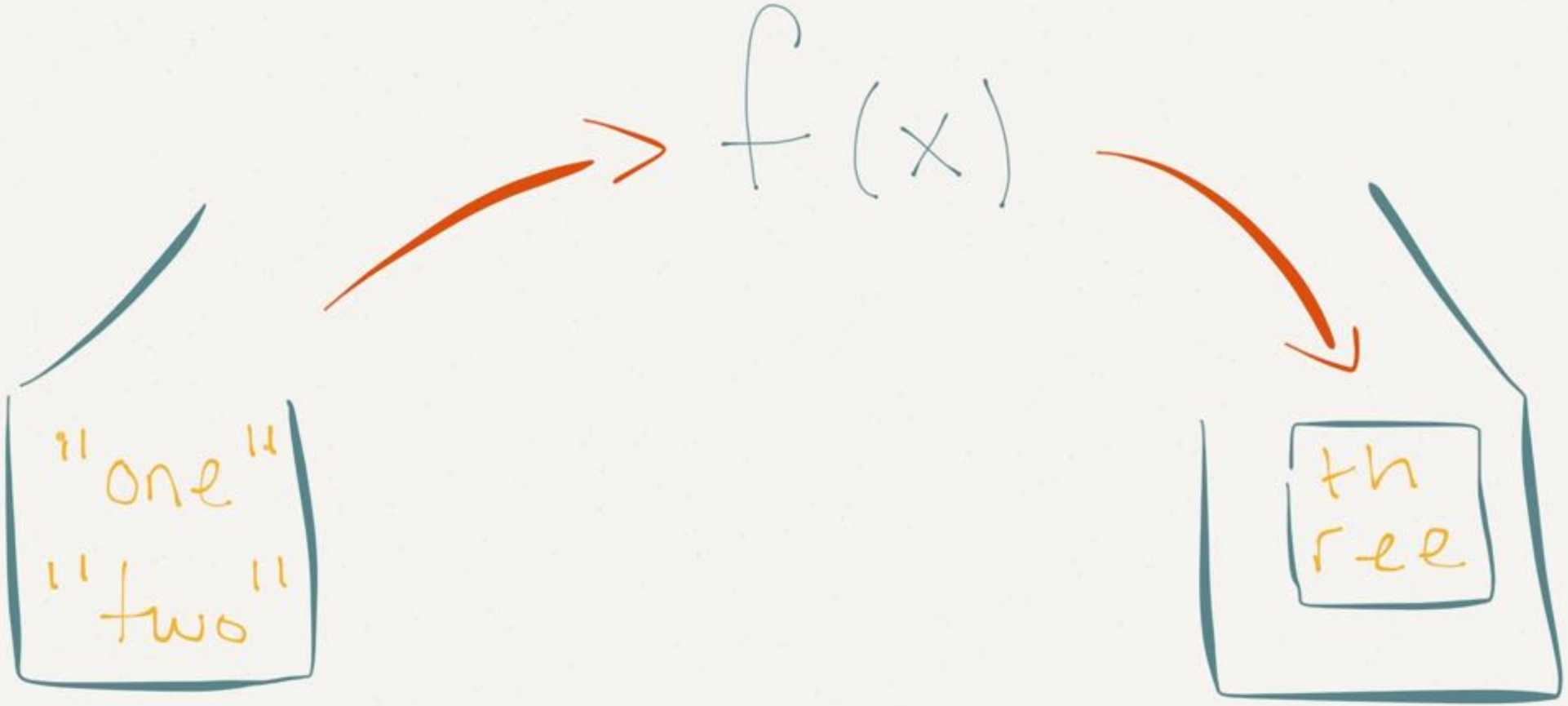
```
func map<T, U>(optional: T?, transform: (T) -> U) -> U?
func apply<T, U>(value: T?,  transform: (T -> U)?) -> U?
```

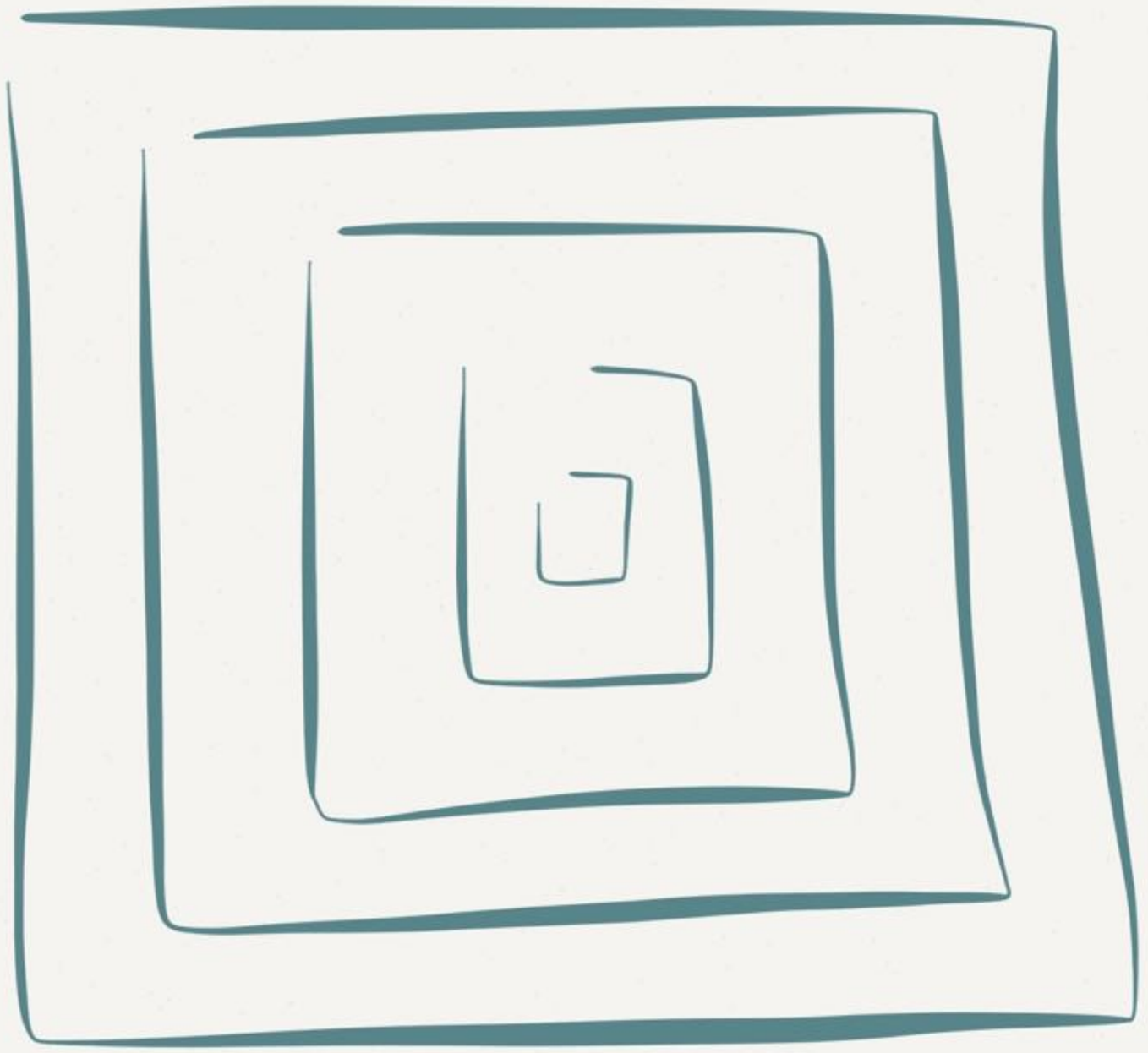
```
func apply<I, O>(transform: ((I) -> O)?, value: I?) -> O? {
    if let v = value, t = transform {
        return Optional.Some(t(v))
    }
    return Optional.None
}
```

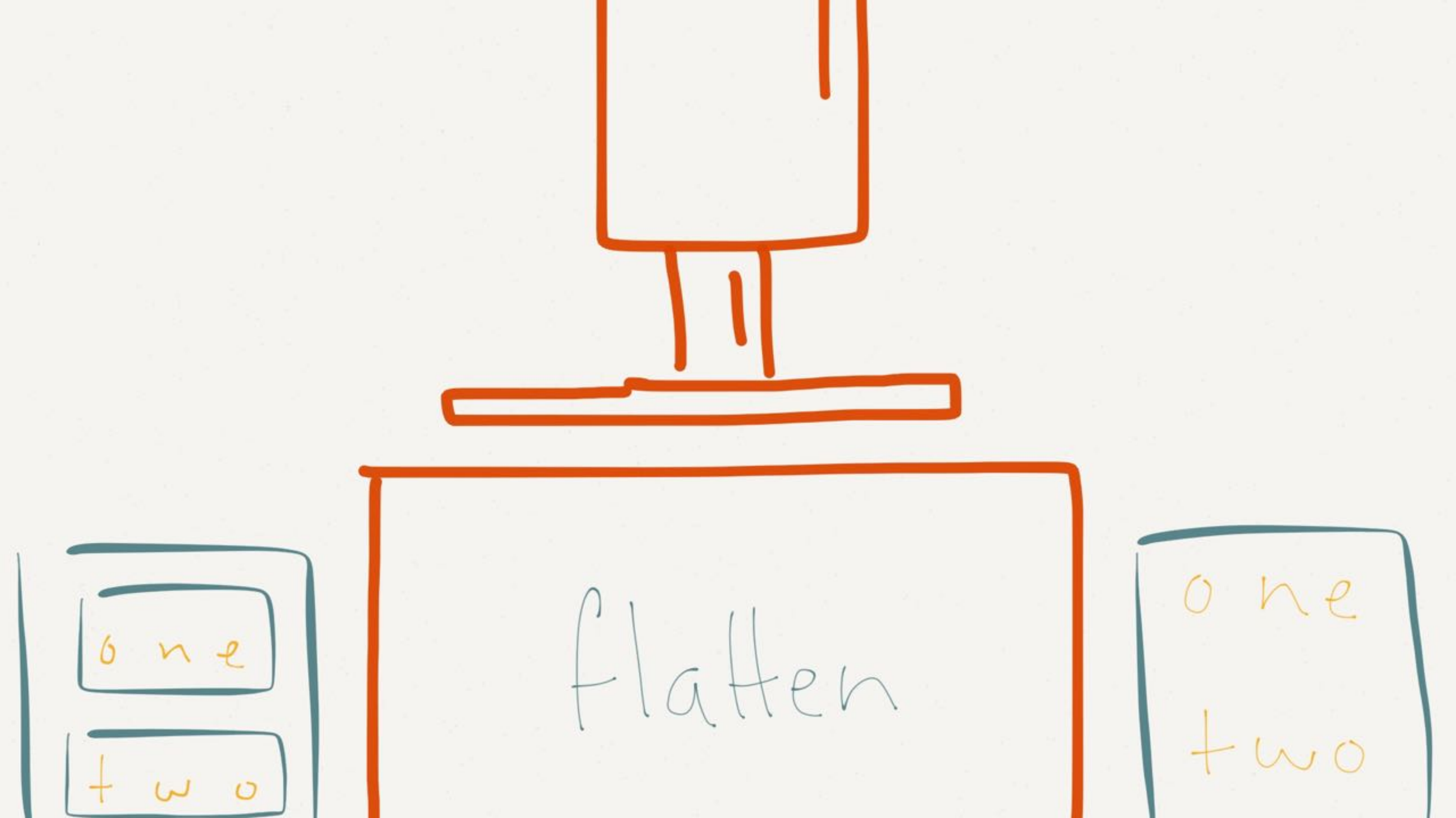
Monads

Mappable 2, The Mappening

```
let strings = ["one", "two", "three"]
let letters = strings.map({ word in Array(word) })
// [["o", "n", "e"], ["t", "w", "o"], ["t", "h", "r", "e", "e"]]
```


f(x)
"one"
"two"
th
ree

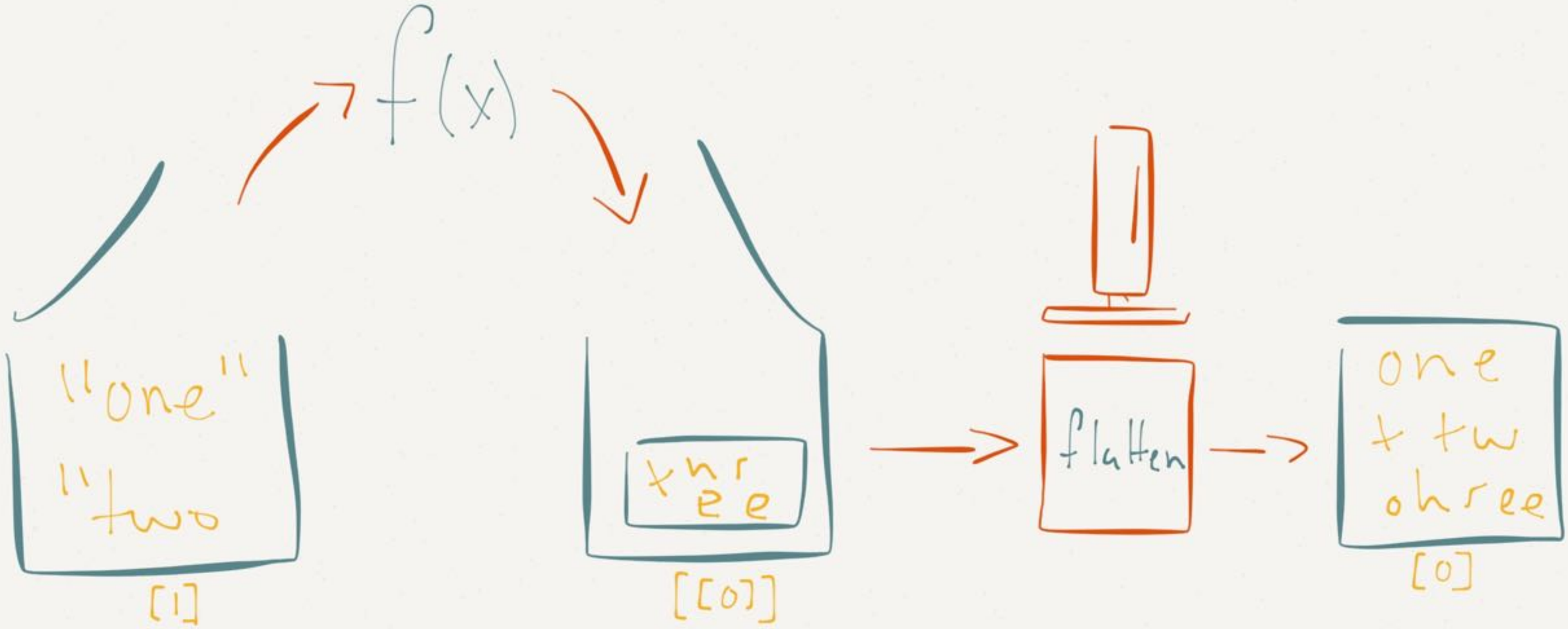



one
two
flatten
one
two

```
func flatten<T>(array: [[T]]) -> [T] {
    var flatArray = [T]()
    for dimension in array {
        flatArray.extend(dimension) // appends elements in dimension to flatArray
    }
    return flatArray
}

flatten(letters)
// ["o", "n", "e", "t", "w", "o", "t", "h", "r", "e", "e"]
```

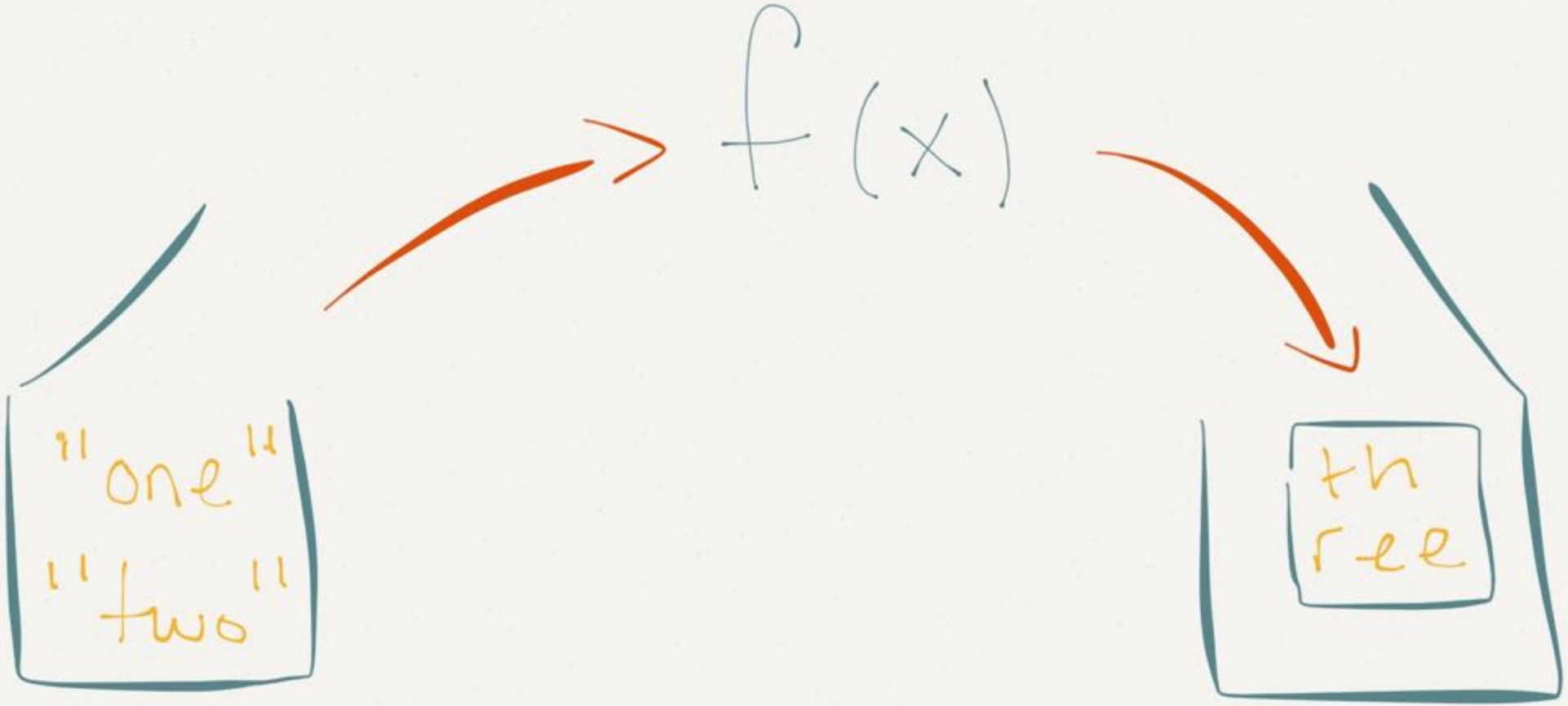
```
let twod = [[1,2,3], [4,5,6], [7,8,9]]
let oned = flatten(twod)
// [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

```
func map<T, U>(x: [T], t: T -> U) -> [U]
func map<T, U>(x: [T], t: T -> [U]) -> [[U]]
```


f(x)
"one"
"two
[1]
thr
ee
[[0]]
flatten
one
two
ohree
[0]

```
func flatMap<T, U>(x: [T], t: T -> [U]) -> [U]
```

```
func flatMap<I, O>(array: [I], transform: I -> [O]) -> [O] {
    return flatten(map(array, transform))
}
```


f(x)
"one"
"two"
th
ree

```
func flatMap<I, O>(array: [I], transform: I -> [O]) -> [O] {
    return flatten(map(array, transform))
}
```

```swift
// Get this out of the way
let fileError = NSError(domain: "com.monads.lol", code: 100, userInfo: nil)
let jsonError = NSError(domain: "com.monads.lol", code: 101, userInfo: nil)

func loadTextFile(path: String) -> Result<String> {
    let fullPath = NSBundle.mainBundle().resourcePath?.stringByAppendingPathComponent(path)
    if let f = fullPath {
        var stringError: NSError?
        let contents = NSString(contentsOfFile: f, encoding: NSUTF8StringEncoding, error: &stringError)
        if let e = stringError {
            return Result.Error(e)
        }
        if let c = contents as? String {
            return Result.Value(Box(c))
        }
    }

    return .Error(fileError)
}
```

```swift
typealias JSON = Dictionary<String, AnyObject>

func parseJSON(json: String) -> Result<JSON> {
    let data = json.dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: false)
    if let d = data {
        var parseError: NSError?
        let dict: AnyObject? = NSJSONSerialization.JSONObjectWithData(d, options: NSJSONReadingOptions(0), error: &parseError)
        if let e = parseError {
            return Result.Error(e)
        }
        if let j = dict as? JSON {
            return Result.Value(Box(j))
        }
    }

    return .Error(jsonError)
}
```

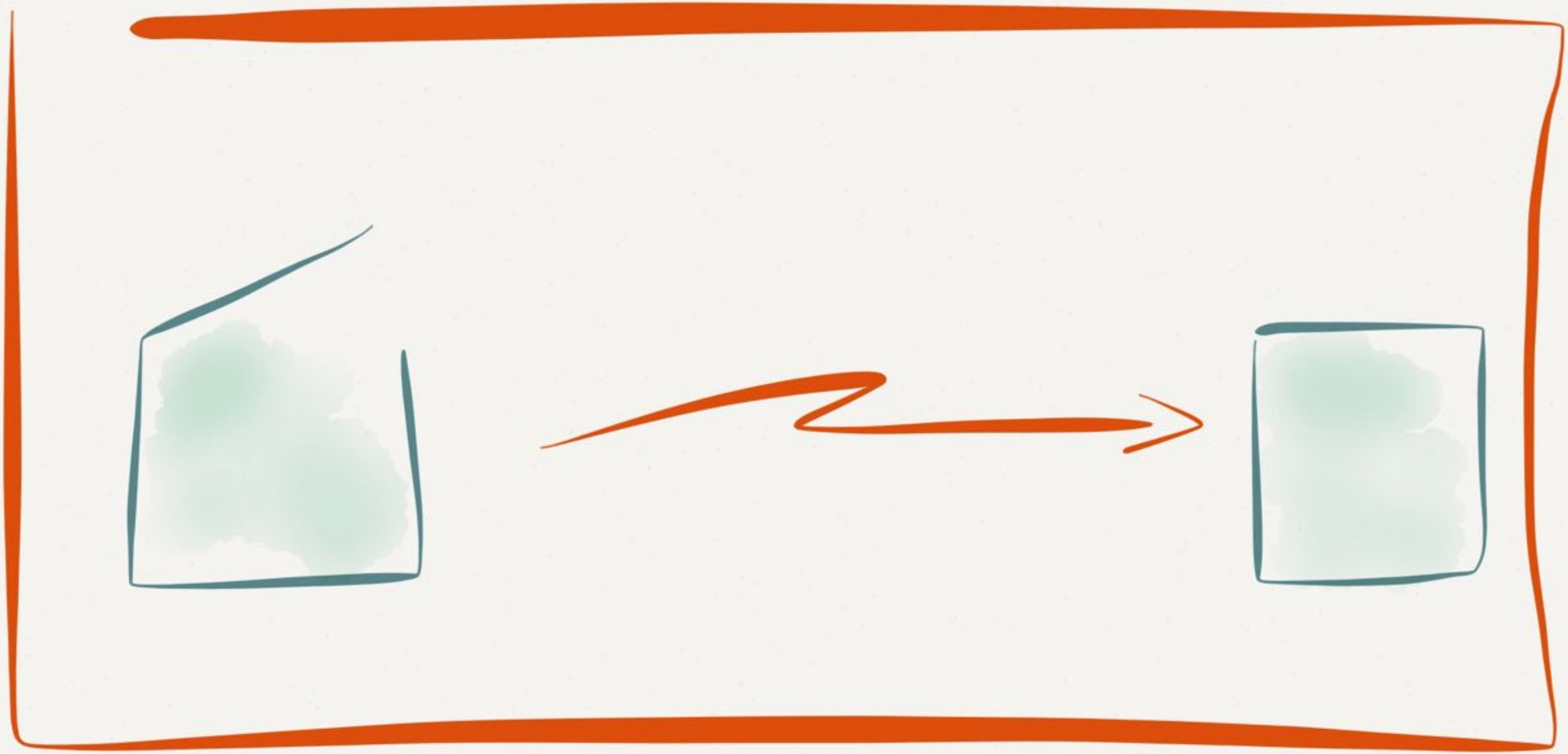
```
func loadTextFile(path: String) -> Result<String> {
    let fullPath = NSBundle.mainBundle().resourcePath?.stringByAppendingPathComponent(path)
    if let f = fullPath {
        var stringError: NSError?
        let contents = NSString(contentsOfFile: f, encoding: NSUTF8StringEncoding, error: &stringError)
        if let e = stringError {
            return Result.Error(e)
        }
        if let c = contents as? String {
            return Result.Value(Box(c))
        }
    }

    return .Error(fileError)
}

func parseJSON(json: String) -> Result<JSON> {
    let data = json.dataUsingEncoding(NSUTF8StringEncoding, allowLossyConversion: false)
    if let d = data {
        var parseError: NSError?
        let dict: AnyObject? = NSJSONSerialization.JSONObjectWithData(d, options: NSJSONReadingOptions(0), error: &parseError)
        if let e = parseError {
            return Result.Error(e)
        }
        if let j = dict as? JSON {
            return Result.Value(Box(j))
        }
    }

    return .Error(jsonError)
}
```

```
let path = "some.json"
let contents: Result<String> = loadTextFile(path)
switch contents {
case .Value(let box):
    let json = parseJSON(box.unbox)
    switch json {
    case .Value(let box2):
        println(box2.unbox)
    case .Error(let error2):
        println(error2)
    }
case .Error(let error):
    println(error)
}

// "[baz: 1234567890, bloop: 0, foo: bar]"
```

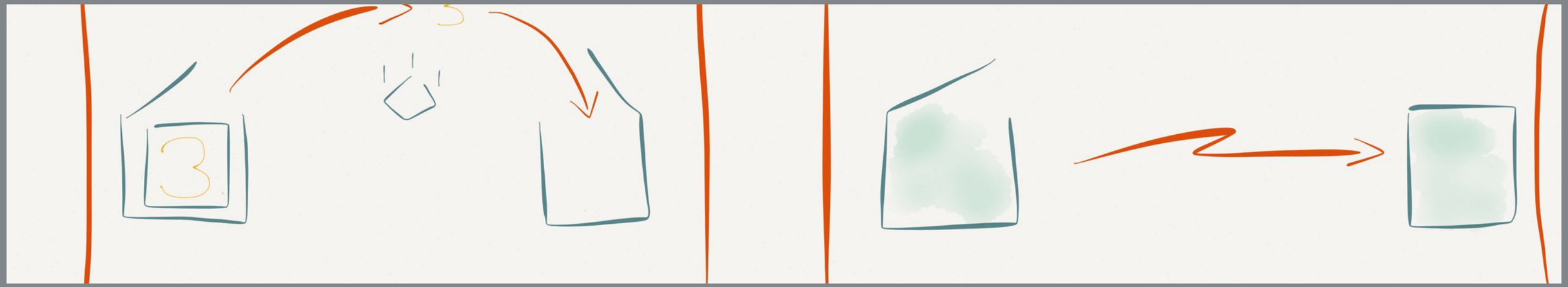
```
func map<I, O>(result: Result<I>, transform: (I) -> O) -> Result<O> {
    switch result {
    case .Value(let v):
        return Result.Value(Box(transform(v.unbox)))
    case .Error(let e):
        return Result.Error(e)
    }
}
```

```
func map<T, U>(x: [T], t: T -> U) -> [U]
func flatMap<T, U>(x: [T], t: T -> [U]) -> [U]
```


3
3



```
func flatten<T>(result: Result<Result<T>>) -> Result<T> {
    switch result {
    case .Value(let box):
        return box.unbox
    case .Error(let error):
        return Result.Error(error)
    }
}
```



```
func flatMap<I, O>(result: Result<I>, transform: (I) -> Result<O>) -> Result<O> {
    return flatten(map(result, transform))
}
```

```
let json = flatMap(loadTextFile("some.json"), parseJSON)
// "[baz: 1234567890, bloop: 0, foo: bar]"
```

```
infix operator >>== { associativity left precedence 150 }
func >>==<T, U>(x: Result<T>, t: T -> Result<U>) -> Result<U> {
    return flatMap(x, t)
}
```

```
let json2 = loadTextFile("some.json") >>== parseJSON
// "[baz: 1234567890, bloop: 0, foo: bar]"
```

```
let path3 = Result.Value(Box("some.json"))
let json3 = path3 >>== loadTextFile >>== parseJSON
// "[baz: 1234567890, bloop: 0, foo: bar]"

let path4 = Result.Value(Box("none.json"))
let json4 = path4 >>== loadTextFile >>== parseJSON
// NSError("The operation couldn’t be completed. No such file or directory")
```

```
path2 >>== loadTextFile >>== parseJSON
```

```
func flatMap<T: Result, U, V>(T<U>, (U) -> T<V>) -> T<V>
func map<T: Result, U, V>(T<U>, (U) -> V)    -> T<V>
```

Functors, Applicative, Monads

```
protocol Functor<T>
protocol Applicative<T>: Functor
protocol Monad<T>: Applicative

    func map<F: Functor,     T, U>(x: F<T>, transform: (T) -> U) ->    F<U> // mappable
  func apply<A: Applicative, T, U>(x: A<T>, transform: ((T) -> U)?) -> A<U> // applicable
func flatMap<M: Monad,       T, U>(x: M<T>, transform: (T) -> U?) ->   M<U> // chainable
```

Further Reading

- How I Learned to Stop Worrying and Love the Functor
- Functors, Applicatives, And Monads In Pictures
- Learn You A Haskell
- Functor and Monad in Swift
- Flattenin' Your Mappenin'
- Deriving Map
- Railway Oriented Programming