ENNIO MASI

SENIOR IOS DEV @ YOOX NET-A-PORTER

DEMYSTIFY FUNCTIONAL PROGRAMMING IN

(OR TRYING TO . . . 🤔)

IN COMPUTER SCIENCE, FP IS A PROGRAMMING PARADIGM – A STYLE OF BUILDING THE STRUCTURE AND ELEMENTS OF COMPUTER PROGRAMS – THAT TREATS COMPUTATION AS THE EVALUATION OF MATHEMATICAL FUNCTIONS AND AVOIDS CHANGING – STATE AND MUTABLE DATA.

Someone on Wikipedia

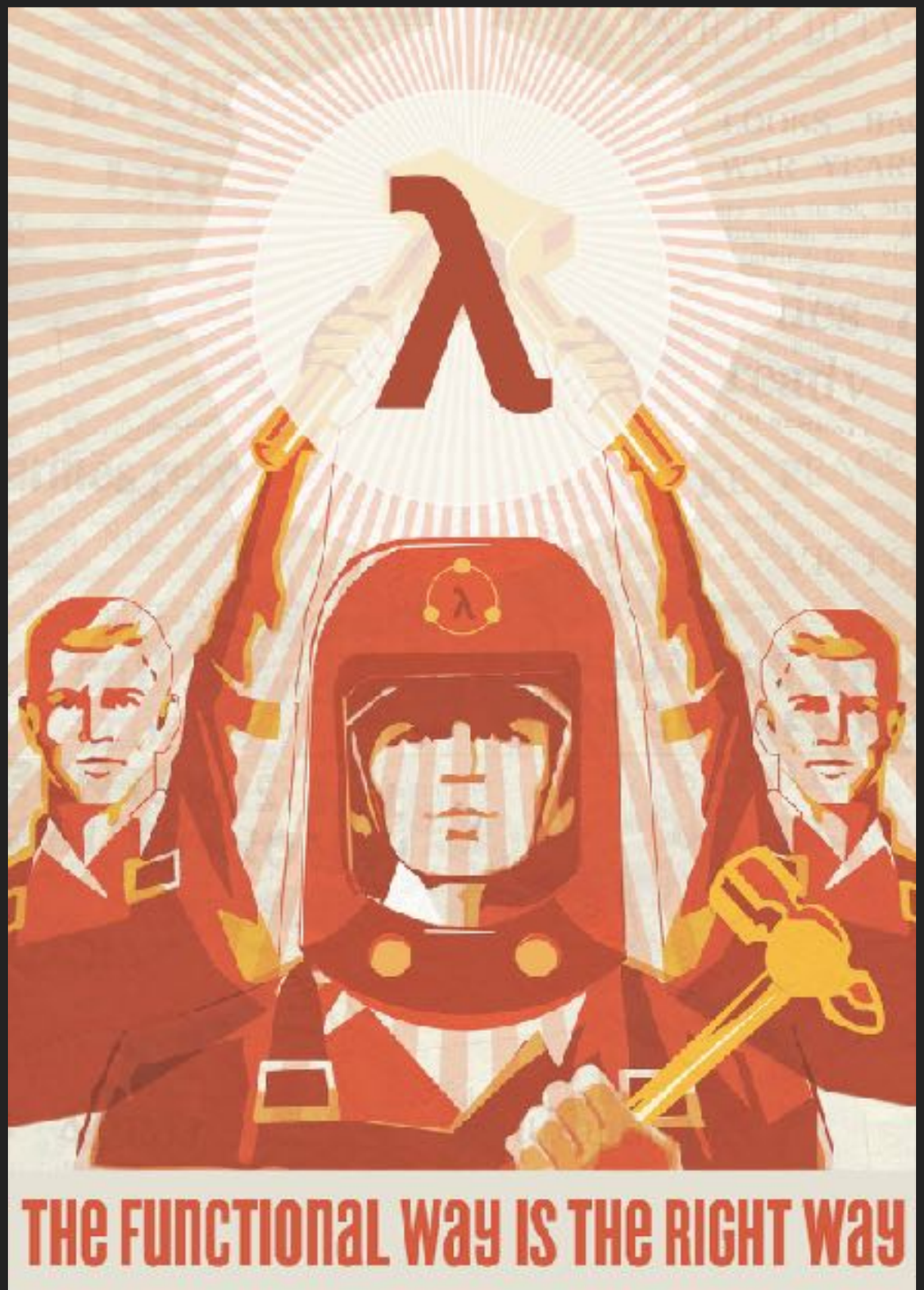
FP HAS ITS ORIGINS IN LAMBDA CALCULUS, A FORMAL SYSTEM DEVELOPED IN THE 1930S TO INVESTIGATE COMPUTABILITY, THE ENTSCHEIDUNGSPROBLEM, FUNCTION DEFINITION, FUNCTION APPLICATION, AND RECURSION. MANY FUNCTIONAL PROGRAMMING LANGUAGES CAN BE VIEWED AS ELABORATIONS ON THE LAMBDA CALCULUS.

Someone else on Wikipedia

FP IS BASED ON CATEGORY THEORY WHICH IS REALLY COMPLEX

FUNCTIONAL PROGRAMMING (HATERS)

FUNCTIONAL PROGRAMMING (LOVERS)



START

FP IS AN APPROACH BASED ON FUNCTION CALLS

```
func doYouLikeSnow(name: String) -> Bool {
    if name == "ennio" {
        return false
    }

    return true
}
```

F(X) —> Y

FP IS AN APP LS

OOP	FP
Single Responsability	functions
Open Closed Principle	functions
Dependecy Inversion	functions
Factory pattern	functions
...	...

FP IS AN APPROACH BASED ON FUNCTION CALLS

```
func triple(value: Int) -> Int {
    return value * 3
}
```

- Write predictable functions
- Code easier to test
- Thread-safe codebase
- Functions chaining

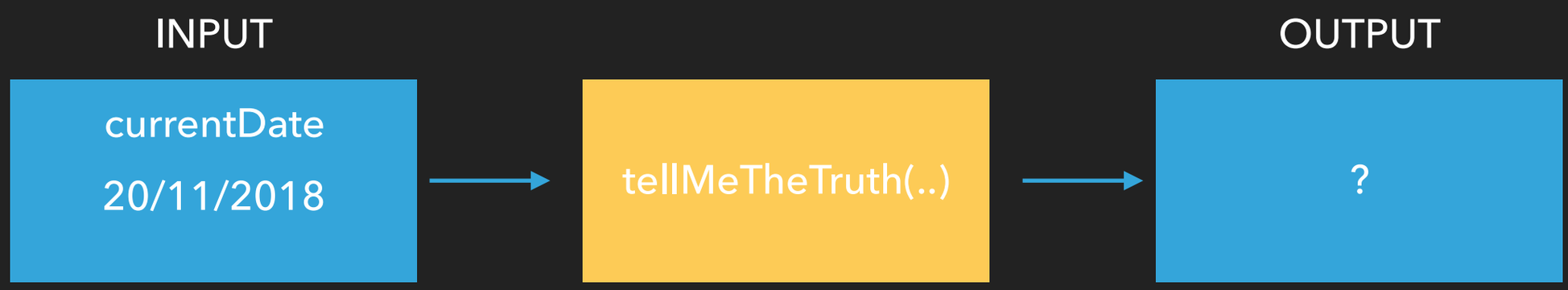
FP IS AN APPROACH BASED ON FUNCTION CALLS

FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

$$F(X) = Y$$

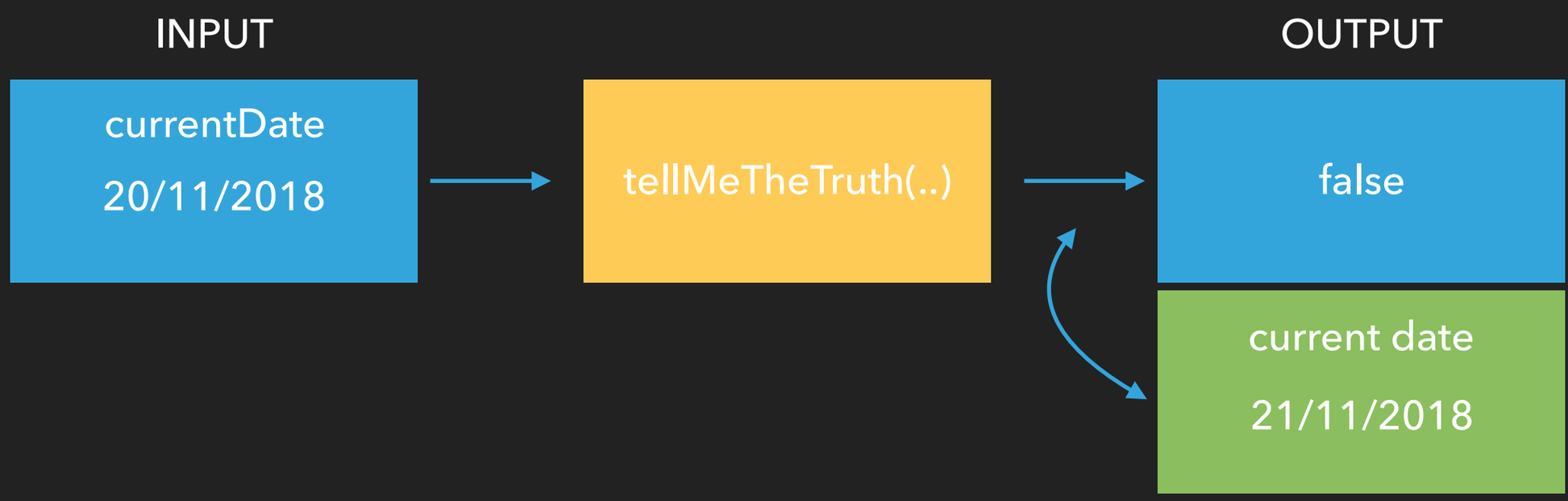
```
func tellMeTheTruth(anInput: SomeData) -> Bool {
    if anInput.currentDate > Date() {
        return true
    }

    return false
}
```



FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

$$F(X) = Y$$

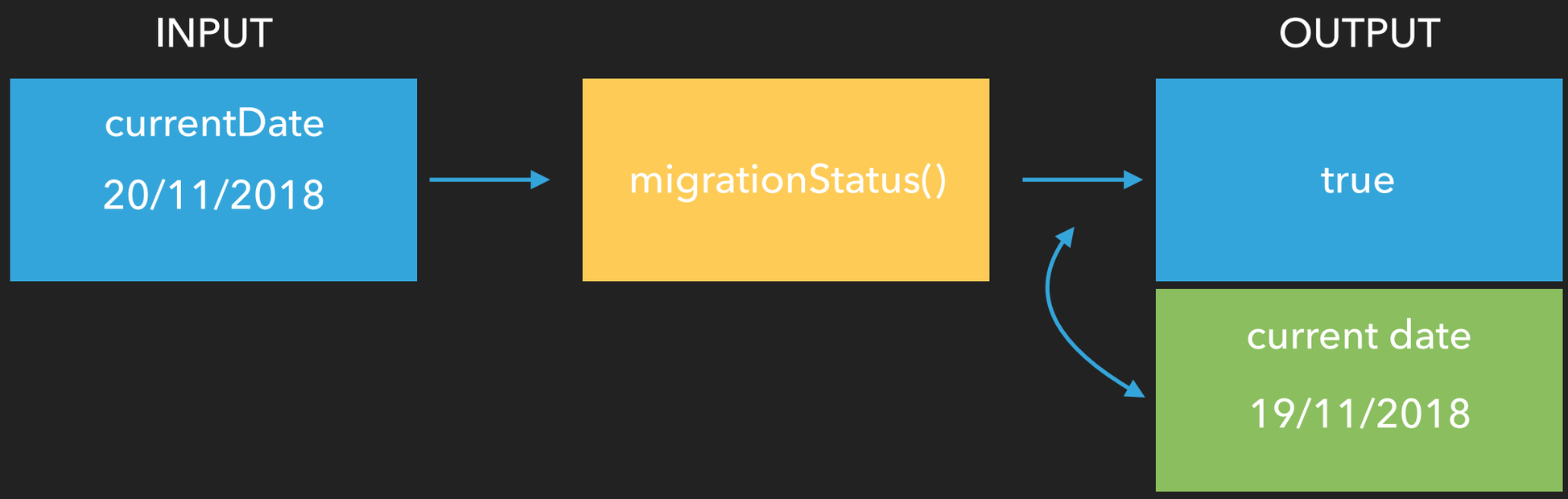
```
func tellMeTheTruth(anInput: SomeData) -> Bool {
    if anInput.currentDate > Date() {
        return true
    }

    return false
}
```



FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

$$F(X) = Y$$

```
func tellMeTheTruth(anInput: SomeData) -> Bool {
    if anInput.currentDate > Date() {
        return true
    }

    return false
}
```



FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

$$F(X) = Y$$

```
func tellMeTheTruth(anInput: SomeData) -> Bool {
    if anInput.currentDate > Date() {
        return true
    }

    return false
}
```

NOT PURE

```
func tellMeTheTruth(anInput: SomeData, compareDate: Date) -> Bool {
    if anInput.currentDate > compareDate {
        return true
    }

    return false
}
```

PURE

NO SIDE EFFECTS, TESTABLE, THREAD SAFE

FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

$$F(X) = Y$$

```
func tellMeTheTruth(anInput: SomeData) -> Bool {
    if anInput.currentDate > Date() {
        return true
    }

    return false
}
```

NOT PURE

```
func tellMeTheTruth(anInput: SomeData, compareDate: Date) -> Bool {
    if anInput.currentDate > compareDate {
        return true
    }

    return false
}
```

PURE

NO SIDE EFFECTS, TESTABLE, THREAD SAFE

FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

$$F(X) = Y$$

```
func tellMeTheTruth(anInput: SomeData) -> Bool {
    if anInput.currentDate > Date() {
        return true
    }

    return false
}
```

NOT PURE

```
func tellMeTheTruth(anInput: SomeData, compareDate: Date) -> Bool {
    if anInput.currentDate > compareDate {
        return true
    }

    return false
}
```

PURE

NO SIDE EFFECTS, TESTABLE, THREAD SAFE

FP IS AN APPROACH BASED ON FUNCTION CALLS

FP IS AN APPROACH BASED ON PURE FUNCTION CALLS

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```
func triple(a: Int) -> Int {
    return a * 3
}
```

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

f(2) –> 2 * 3 = 6

g(6) –> 6 * 6 = 36

```
1  func triple(a: Int) -> Int { return a * 3 }          (2 times)
2  func square(a: Int) -> Int { return a * a}           (2 times)
3
4  let t = triple(a: 2)                                 6
5  let result = square(a: t)                            36
6
7  square(a: triple(a: 2))                              36
8
```

g(f(2)) ➡️ f(2) = 6 ➡️ g(6) = 36

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```
func triple(a: Int) -> Int {
    return a * 3
}
```

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

HIGH ORDER FUNCTIONS

```
let array = [1, 2, 3, 4, 5, 6]
array.map{ return $0 * 3 }
```

FUNCTIONS ARE FIRST CLASS CITIZENS

```
typealias AwesomeFunction = (Int) -> ()

let myAwesomeFunction = { input in
    print(input)
}

myAwesomeFunction(3)
```

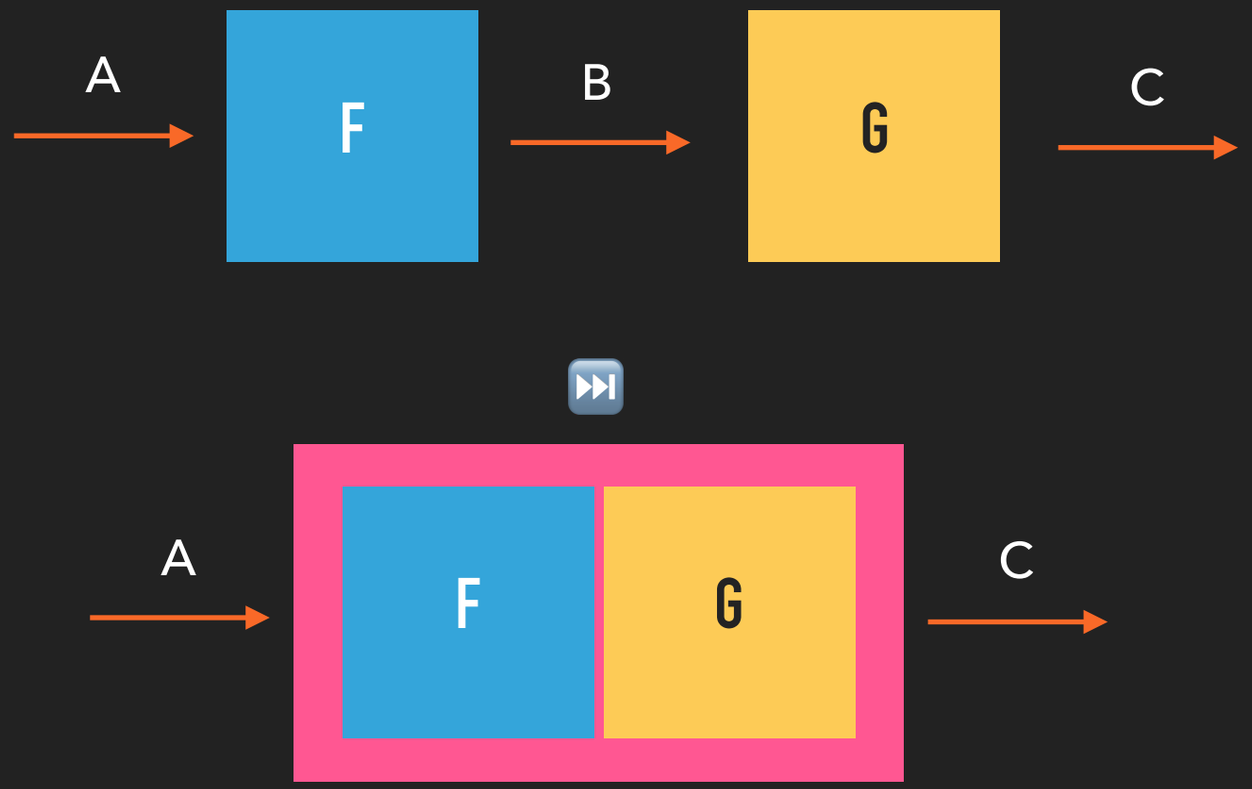
SWIFT SUPPORTS FP

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```
func triple(a: Int) -> Int {
    return a * 3
}
```

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

COMPOSITION



FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```swift
func triple(a: Int) -> Int {
    return a * 3
}
```

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

COMPOSITION

```swift
precedencegroup FunctionComposition {
    associativity: left
}

infix operator ⏭: FunctionComposition

func ⏭<A, B, C>(f: @escaping (A) -> B, g: @escaping (B) -> C) -> ((A) -> C) {
    return { input in
        return g(f(input))
    }
}
```

```swift
(triple ⏭ square)(2)
```

```
36
```

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```
precedencegroup FunctionComposition {
    associativity: left
}

infix operator ⏭: FunctionComposition

func ⏭<A, B, C>(f: @escaping (A) -> B, g: @escaping (B) -> C) -> ((A) -> C) {
    return { input in
        return g(f(input))
    }
}
```

F: A –>B G: B –>C

```
(triple ⏭ square)(2)
```

36

F: INT –>INT G: INT –>INT

```
(triple ⏭ square ⏭ String.init)(2)
```

"36"

F: INT –>INT G: INT –>INT H: INT –>STR

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```
(triple ⏭ square ⏭ String.init)(2)                    "36"
```

```
1  func triple(a: Int) -> Int { return a * 3 }        (3 times)
2  func square(a: Int) -> Int { return a * a}         (3 times)
3
4  let t = triple(a: 2)                               6
5  let result = square(a: t)                          36
6
7  square(a: triple(a: 2))                            36
8  String.init(square(a: triple(a: 2)))               "36"
```

FP IS AN APPROACH BASED ON COMPOSITION OF PURE FUNCTION CALLS

```
precedencegroup FunctionComposition {
    associativity: left
}

infix operator ⏭: FunctionComposition

func ⏭<A, B, C>(f: @escaping (A) -> B, g: @escaping (B) -> C) -> ((A) -> C) {
    return { input in
        return g(f(input))
    }
}
```

```
(triple ⏭ square ⏭ String.init)(2)                                    "36"
```

- Write predictable functions
- Code easier to test
- Thread-safe codebase
- Features chaining

CURRYING IS A WAY TO BREAKDOWN FUNCTIONS COMPLEXITY 👍🏼

- Think to a function with N parameters as a function with N times 1 parameter 🙄
- A curried function is executed only when all the parameters are set

F(X, Y) —> Z

F(X)(Y) —> Z

```
typealias CurryFunction<A, B, C> = (A, B) -> C
typealias CurryOutputFunction<A, B, C> = (A) -> ((B) -> C)

func curry<A, B, C>(f: @escaping CurryFunction<A, B, C>) -> CurryOutputFunction<A, B, C> {
    return { a in { b in f(a, b) } }
}
```

FOUNDATIONS: IMMUTABILITY

IMPERATIVE VS FP

```swift
func sum(from data: [Int]) -> Int {
    var s = 0

    for i in 0..<data.count {
        s = s + data[i]
    }

    return s
}

var data = [1, 2, 3, 4, 5, 6]
sum(from: data)
```

```
0

(6 times)

21


[1, 2, 3, 4, 5, 6]
21
```

S + DATA[I] IS “ILLEGAL” IN FP 😈

PURE FUNCTIONS MUST BE STATELESS

OOP (IMPERATIVE) VS FP

TAIL RECURSION

```
func sum2(from data: [Int], acc: Int) -> Int {
    if let value = data.first {
        return acc + sum2(from: Array(data.dropFirst()), acc: value)   (6 times)
    } else {
        return acc                                                     6
    }
}
sum2(from: data, acc: 0)                                               21
```

TAIL RECURSION IS NOT OPTIMISED IN SWIFT 🙈

SWIFT PARTIALLY SUPPORTS FP

ADVANTAGES OF IMMUTABILITY

- Predictable outputs
- Thread-safe
- Chaining of functions

DISADVANTAGES OF IMMUTABILITY

- Performances
- Complex code base

"A monad is just a monoid in the category of endofunctors"

Someone on Internet


HEY BABY
I KNOW WHAT
A MONAD IS

SWIFT OPTIONAL

```
public enum Optional<Wrapped> : ExpressibleByNilLiteral {
  // The compiler has special knowledge of Optional<Wrapped>, including the fact
  // that it is an `enum` with cases named `none` and `some`.

  /// The absence of a value.
  ///
  /// In code, the absence of a value is typically written using the `nil`
  /// literal rather than the explicit `.none` enumeration case.
  case none

  /// The presence of a value, stored as `Wrapped`.
  case some(Wrapped)

  /// Creates an instance that stores the given value.
  @_inlineable // FIXME(sil-serialize-all)
  @_transparent
  public init(_ some: Wrapped) { self = .some(some) }
```

OPTIONAL IS A “CONTEXT” THAT CAN INCLUDE A VALUE OR NOT

FUNCTORS

- SINCE OPTIONAL IS A “CONTEXT”, WE NEED A WAY TO EXTRACT THE POSSIBLE VALUE FROM IT

- MAP 😍

```
let optional1: Int? = nil
optional1.map { return $0 * 2 }
```

```
nil
nil
```

```
let optional2: Int? = 3
let mappedOptional2 = optional2.map{ return $0 * 2 }
print(mappedOptional2)
```

⚠ Expression implicitly coerced from 'Int?' to Any

```
3
(2 times)
"Optional(6)\n"
```

FUNCTORS

- SINCE OPTIONAL IS A “CONTEXT”, WE NEED A WAY TO EXTRACT THE POSSIBLE VALUE FROM IT

- MAP 😍 [<$>, <!>]

```
let optional1: Int? = nil
optional1.map { return $0 * 2 }
```

```
nil
nil
```

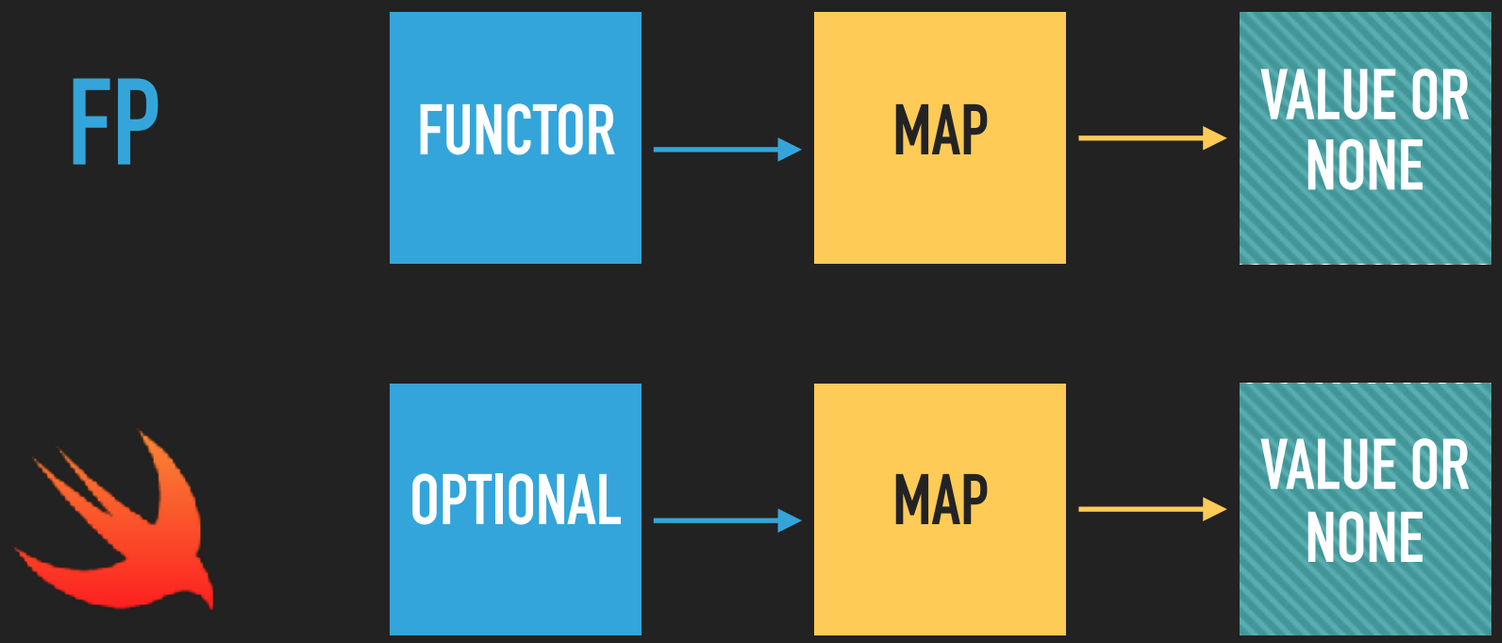
```
let optional2: Int? = 3
let mappedOptional2 = optional2.map{ return $0 * 2 }
print(mappedOptional2)
```

⚠ Expression implicitly coerced from 'Int?' to Any

```
3
(2 times)
"Optional(6)\n"
```

FUNCTORS

A FUNCTOR IS A TYPE THAT CONTAINS A VALUE AND PROVIDES A MAP FUNCTION AS INTERFACE



HINT: OPTIONAL IS A FUNCTOR

FUNCTORS

A FUNCTOR IS A TYPE THAT CONTAINS A VALUE AND PROVIDES A MAP FUNCTION AS INTERFACE

- Identity Law
- Composition Law

```
[1, 2, 3, 4, 5, 6].map { $0 }
```

```
93 print([1, 2, 3, 4, 5, 6].map { square(triple($0)) })
94 print([1, 2, 3, 4, 5, 6].map { triple($0) }.map { square($0) })
95
```

```
[9, 36, 81, 144, 225, 324]
[9, 36, 81, 144, 225, 324]
```

HINT: OPTIONAL IS A FUNCTOR

MONADS

A MONAD IS JUST A MONOID IN THE BLA BLA BLA ...

- Well a monad is *just* a type on which we can apply flatMap (bind)

FUNCTIONS CHAINING (ON VALUES)

- flatMap [>>=]

HINT: OPTIONAL IS A MONAD 👍🏼

MONADS IN PRACTICE

Well a monad is *just* a type on which we can apply flatMap (bind)

```
enum Result<T> {
    case success(T)
    case failure(NSError)
}

extension Result {
    func bind<U>(f: ((T) -> Result<U>)) -> Result<U> {
        switch self {
        case let .success(value):
            return f(value)
        case let .failure(error):
            return .failure(error)
        }
    }
}
```

MONADS

```
enum Result<T> {
    case success(T)
    case failure(NSError)
}
```

```
func evaluateMyString(inputString: String) -> Result<String> {
    print("1 evaluateMyString")
    if inputString.count % 2 == 0 {
        return Result.success(inputString)
    } else {
        return Result.failure(NSError(domain: "aDomain", code: 1, userInfo: nil))
    }
}
```

```
> Result<U> {
    return .failure(error)
```

```
func doSomethingWithThatString(inputString: String) -> Result<Bool> {
    print("2 doSomethingWithThatString")
    if inputString == "Snow" { return Result.success(true) }

    return Result.failure(NSError(domain: "aDomain", code: 2, userInfo: nil))
}
```

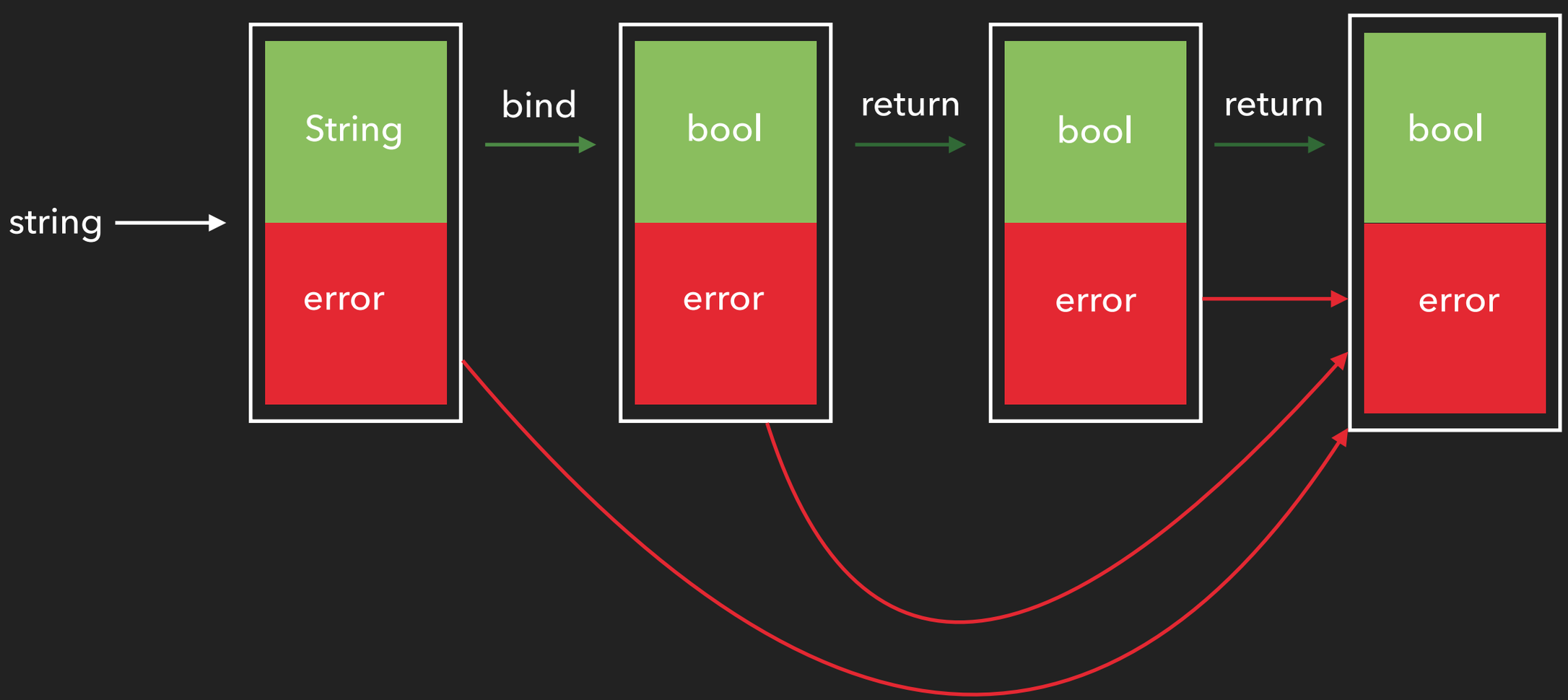
```
func generalFunction(inputString: String) -> Result<Bool> {
    return evaluateMyString(inputString: inputString).bind { doSomethingWithThatString(inputString: $0)}
}
```

```swift
func evaluateMyString(inputString: String) -> Result<String> {
    print("1 evaluateMyString")
    if inputString.count % 2 == 0 {
        return Result.success(inputString)
    } else {
        return Result.failure(NSError(domain: "aDomain", code: 1, userInfo: nil))
    }
}
```

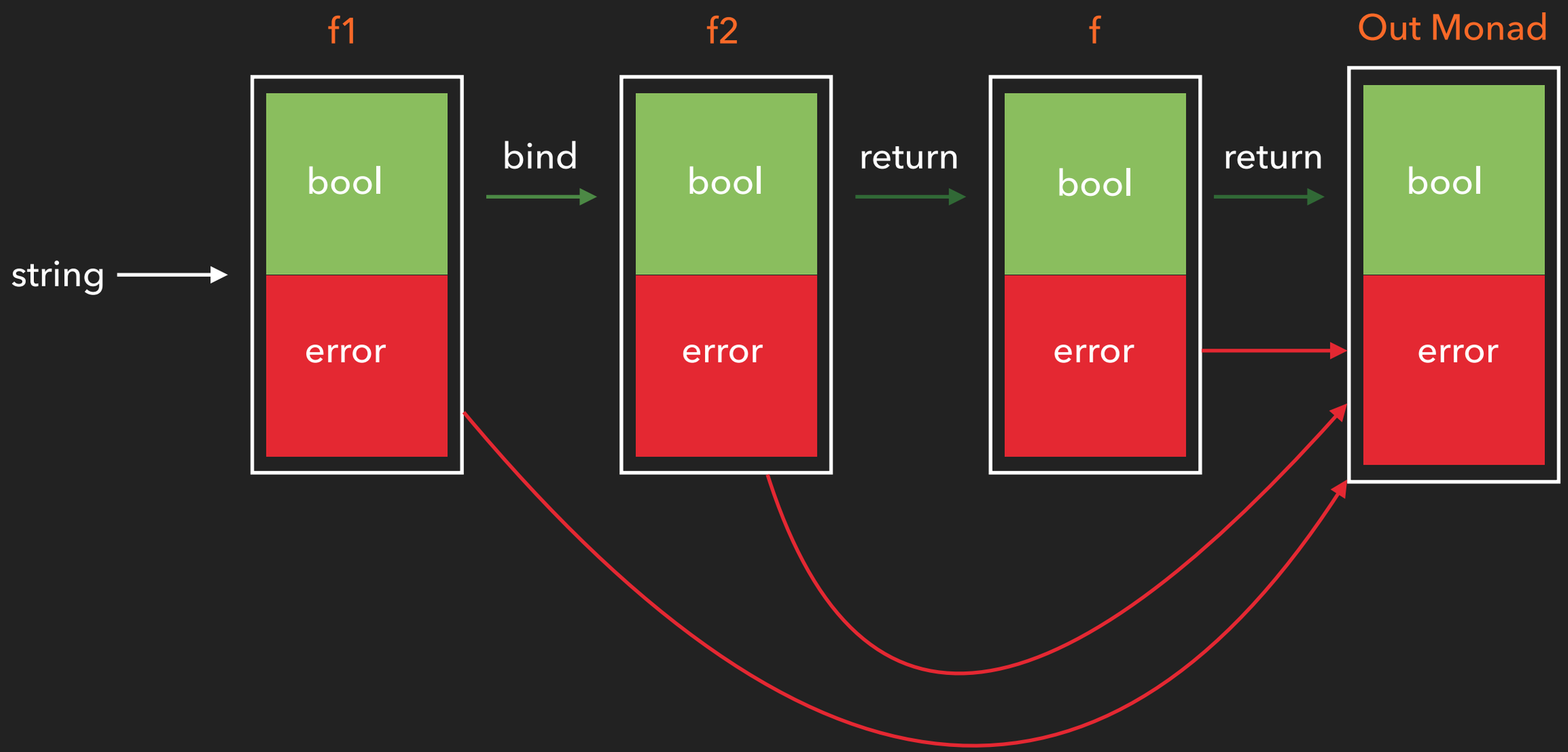
```swift
func doSomethingWithThatString(inputString: String) -> Result<Bool> {
    print("2 doSomethingWithThatString")
    if inputString == "Snow" { return Result.success(true) }

    return Result.failure(NSError(domain: "aDomain", code: 2, userInfo: nil))
}
```

```swift
func generalFunction(inputString: String) -> Result<Bool> {
    return evaluateMyString(inputString: inputString).bind { doSomethingWithThatString(inputString: $0)}
}
```

f1 f2 f Out Monad



FOUNDATIONS: FEATURES CHAINING



FEATURES CHAINING (ON VALUES)
VS
ERROR MANAGEMENT

PLAYGROUND TIME 🔥

https://github.com/ennioma/FPTalk-Playground