Combinators

NSSpain 2018

Daniel H Steinberg

dimsumthinking.com and editorscut.com

Note: I never do this

"[Combinators are] functions that, when seen as lambda terms, contain no free variables."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"[Combinators are] functions that, when seen as lambda terms, contain no free variables."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"[Combinators are] functions that, when seen as lambda terms, contain no free variables."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"[Combinators are] functions that, when seen as lambda terms, contain no free variables."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"[Combinators are] functions that, when seen as lambda terms, contain no free variables."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

Combinators were created…

Combinators were created…

by someone on Swift Evolution two years ago

Combinators were created…

Combinators were created…

in Haskell, 'cause everything good about Swift was

Combinators were created…

Combinators were created…

nearly 100 years ago by Moses Shönfinkel

Combinators were created…

Combinators were created…

independently by Haskell Curry ninety years ago

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

Combinator

Parser - Combinator

Y - Combinator

Y - Combinator

A fixed-point combinator

Combinator

Sets

Sets

Swift Standard Libary

```
let primes: Set = [2, 3, 5, 7]
```

```
let primes: Set = [2, 3, 5, 7]


primes.contains(4)
```

```
let primes: Set = [2, 3, 5, 7]

primes.contains(4)

                    false
```

```
let primes: Set = [2, 3, 5, 7]

primes.contains(5)
```

```
let primes: Set = [2, 3, 5, 7]

primes.contains(5)

true
```

```
let primes: Set = [2, 3, 5, 7]
```

```swift
let primes: Set = [2, 3, 5, 7]

primes.map{x in x * 10}
```

```
let primes: Set = [2, 3, 5, 7]

primes.map{x in x * 10}

{70, 20, 50, 30}
```

```
let primes: Set = [2, 3, 5, 7]


primes.map{x in x * 10}
```

```swift
let primes: Set = [2, 3, 5, 7]


let y = 10
primes.map{x in x * y}
```

"[Combinators are] functions that, when seen as lambda terms, contain no free variables."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

```
let primes: Set = [2, 3, 5, 7]


primes.map{x in x * 10}
```

Combinator

```
let primes: Set = [2, 3, 5, 7]


let y = 10
primes.map{x in x * y}
```

Not a Combinator

```
let primes: Set = [2, 3, 5, 7]
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
```

```swift
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
```

```
{3, 5, 7}
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
```

```
{3, 7, 1, 2, 5, 9}
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
primes.symmetricDifference(odds)
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
primes.symmetricDifference(odds)
```

{1, 2, 9}

```swift
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
primes.symmetricDifference(odds)
```

```swift
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
primes.symmetricDifference(odds)
```

Combinators

"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

```swift
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

primes.intersection(odds)
primes.union(odds)
primes.symmetricDifference(odds)
```

Combinators

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]
```

```
let primes: Set = [2, 3, 5, 7]
let odds: Set = [1, 3, 5, 7, 9]

let evens: Set = […, -2, 0, 2, …]
```

Infinite Sets

```
struct IntSet {

}
```

```
struct IntSet {
    let contains: (Int) -> Bool
}
```

```swift
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet
```

```
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet(contains: ?)
```

```
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet(contains: {x in
    x % 2 == 0
})
```

```swift
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet(contains: {x in
    x % 2 == 0
})
```

```
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet{x in
                   x % 2 == 0}
```

```swift
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet{x in
                    x % 2 == 0}

     evens.contains(-400)
```

```
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet{x in
                x % 2 == 0}
```

```
evens.contains(-400)
```

```
true
```

```
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet{x in
                   x % 2 == 0}
```

```
evens.contains(1013)
```

```
struct IntSet {
    let contains: (Int) -> Bool
}

let evens = IntSet{x in
                    x % 2 == 0}
```

```
evens.contains(1013)
```

false

```
let twoThreeFour
```

```
let twoThreeFour =
          IntSet(withRangeFrom: 2,
                 to: 4)
```

```swift
struct IntSet {
    let contains: (Int) -> Bool
}
```

```
extension IntSet {
    init(withRangeFrom lower: Int,
         to upper: Int) {
        contains = {x in
            (x >= lower) && (x <= upper)
        }
    }
}
```

```
extension IntSet {
    init(withRangeFrom lower: Int,
         to upper: Int) {
        contains = {x in
            (x >= lower) && (x <= upper)
        }
    }
}
```

```
extension IntSet {
    init(withRangeFrom lower: Int,
         to upper: Int) {
        contains = {x in
            (x >= lower) && (x <= upper)
        }
    }
}
```

```
extension IntSet {
    init(withRangeFrom lower: Int,
         to upper: Int) {
        contains = {x in
            (x >= lower) && (x <= upper)
        }
    }
}
```

```
let twoThreeFour =
           IntSet(withRangeFrom: 2,
                  to: 4)

let primes
```

```
let twoThreeFour =
            IntSet(withRangeFrom: 2,
                             to: 4)

let primes = IntSet(2, 3, 5, 7)
```

```
struct IntSet {
    let contains: (Int) -> Bool
}
```

```
extension IntSet {
    init(_ elements: Int ...) {
        contains = {x in
            elements.contains(x)
        }
    }
}
```

```
extension IntSet {
    init(_ elements: Int ...) {
        contains = {x in
            elements.contains(x)
        }
    }
}
```

```
let evens = IntSet{x in
                     x % 2 == 0}
let twoThreeFour =
         IntSet(withRangeFrom: 2,
                to: 4)

let primes = IntSet(2, 3, 5, 7)
```

```
let evens = IntSet{x in
                     x % 2 == 0}
let twoThreeFour =
         IntSet(withRangeFrom: 2,
                to: 4)

let primes = IntSet(2, 3, 5, 7)

let emptySet = IntSet()
```

```
let evens = IntSet{x in
                     x % 2 == 0}
let twoThreeFour =
         IntSet(withRangeFrom: 2,
                to: 4)

let primes = IntSet(2, 3, 5, 7)

let emptySet = IntSet()

let universalSet = IntSet{_ in
                            return true}
```

Combinators

```
let addSeven = twoThreeFour.add(7)
```

```
let addSeven = twoThreeFour.add(7)
```

```
let addSeven = twoThreeFour.add(7)
```

```
let addSeven = twoThreeFour.add(7)
```

```
[2, 3, 4, 7]
```

```
let addSeven = twoThreeFour.add(7)
let removeSeven = addSeven.remove(7)
```

```
let addSeven = twoThreeFour.add(7)
let removeSeven = addSeven.remove(7)
```

```
[2, 3, 4]
```

```
let addSeven = twoThreeFour.add(7)
let removeSeven = addSeven.remove(7)
let addSevenAgain = removeSeven.add(7)
```

```
let addSeven = twoThreeFour.add(7)
let removeSeven = addSeven.remove(7)
let addSevenAgain = removeSeven.add(7)
```

```
[2, 3, 4, 7]
```

```
extension IntSet {
    func add(_ element: Int) -> IntSet {
        return IntSet{x in
            self.contains(x) || x == element
        }
    }
}
```

```
extension IntSet {
    func add(_ element: Int) -> IntSet {
        return IntSet{x in
            self.contains(x) || x == element
        }
    }
}
```

```
extension IntSet {
    func add(_ element: Int) -> IntSet {
        return IntSet{x in
            self.contains(x) || x == element
        }
    }
}
```

```swift
extension IntSet {
    func add(_ element: Int) -> IntSet {
        return IntSet{x in
            self.contains(x) || x == element
        }
    }
    func remove(_ element: Int) -> IntSet {
        return IntSet{ x in
            self.contains(x) && x != element
        }
    }
}
```

```swift
extension IntSet {
    func add(_ element: Int) -> IntSet {
        return IntSet{x in
            self.contains(x) || x == element
        }
    }
    func remove(_ element: Int) -> IntSet {
        return IntSet{ x in
            self.contains(x) && x != element
        }
    }
}
```

```
extension IntSet {
    func add(_ element: Int) -> IntSet {
        return IntSet{x in
            self.contains(x) || x == element
        }
    }
    func remove(_ element: Int) -> IntSet {
        return IntSet{ x in
            self.contains(x) && x != element
        }
    }
}
```

```
let addSeven = twoThreeFour.add(7)
let removeSeven = addSeven.remove(7)
let addSevenAgain = removeSeven.add(7)
```

```
twoThreeFour.union(primes)

twoThreeFour.intersection(primes)

twoThreeFour.minus(primes)

twoThreeFour
    .symmetricDifference(with: primes)
```

```
extension IntSet {
    func union(_ otherSet: IntSet)
                            -> IntSet {
        return IntSet{x in
            (self.contains(x) ||
             otherSet.contains(x))}
    }
}
```

```
extension IntSet {
    func union(_ otherSet: IntSet)
                           -> IntSet {
        return IntSet{x in
            (self.contains(x) ||
             otherSet.contains(x))}
    }
}
```

```
extension IntSet {
    func union(_ otherSet: IntSet)
                             -> IntSet {
        return IntSet{x in
            (self.contains(x) ||
             otherSet.contains(x)) }
    }
}
```

```
extension IntSet {
    func intersection(_ otherSet: IntSet)
                                    -> IntSet {
        return IntSet{ x in
            (self.contains(x) &&
             otherSet.contains(x))}
    }
}
```

```
extension IntSet {
    func intersection(_ otherSet: IntSet)
                                    -> IntSet {
        return IntSet{ x in
            (self.contains(x) &&
             otherSet.contains(x))}
    }
}
```

```
extension IntSet {
    var complement: IntSet {
        return IntSet{x in
                        !self.contains(x)}
    }
}
```

```
extension IntSet {
    var complement: IntSet {
        return IntSet{x in
                          !self.contains(x)}
    }
}
```

```
IntSet -> IntSet
```

Combinators

```
IntSet -> IntSet
```

```swift
struct IntSet {
    let contains: (Int) -> Bool
}
```

```
twoThreeFour.union(primes)

twoThreeFour.intersection(primes)

twoThreeFour.complement

twoThreeFour.add(7)

twoThreeFour.remove(2)
```

State

```
struct State<S, A> {
    let run: (S) -> (A, S)
}
```

```
struct State<S, A> {
    let run: (S) -> (A, S)
}
```

```
struct State<S, A> {
    let run: (S) -> (A, S)
}
```

```
typealias Rand<A> = State<RNG, A>
```

Functional Programming in Scala by Paul Chiusano and Runar Bjarnason

```
typealias Rand<A> = State<RNG, A>
```

```
typealias Rand<A> = State<RNG, A>
```

```swift
struct RNG {
    let seed: Int

    func next() -> (Int, RNG) {
        let newSeed = (seed * A + C) % M
        let nextRNG = RNG(seed: newSeed)
        return (newSeed, nextRNG)
    }
}
```

```swift
struct RNG {
    let seed: Int

    func next() -> (Int, RNG) {
        let newSeed = (seed * A + C) % M
        let nextRNG = RNG(seed: newSeed)
        return (newSeed, nextRNG)
    }
}
```

```swift
struct RNG {
    let seed: Int

    func next() -> (Int, RNG) {
        let newSeed = (seed * A + C) % M
        let nextRNG = RNG(seed: newSeed)
        return (newSeed, nextRNG)
    }
}
```

```
struct State<S, A> {
    let run: (S) -> (A, S)
}

typealias Rand<A> = State<RNG, A>
```

```
struct State<S, A> {
    let run: (S) -> (A, S)
}

typealias Rand<A> = State<RNG, A>

run: (RNG) -> (Int, RNG)
```

```
let intGenerator
    = Rand<Int>{rng in rng.next()}
```

```
let intGenerator
    = Rand<Int>{rng in rng.next()}
```

```
let intGenerator
    = Rand<Int>{rng in rng.next()}

func next() -> (Int, RNG) { //..
      return (newSeed, nextRNG)
}
```

```
let (int1, intRNG1)
    =  intGenerator.run(initialRNG)
```

```
let (int1, intRNG1)
    =  intGenerator.run(initialRNG)
```

```
let (int1, intRNG1)
     =  intGenerator.run(initialRNG)
```

```
let (int1, intRNG1)
    =  intGenerator.run(initialRNG)
```

3102

```
let (int2, intRNG2)
    =  intGenerator.run(intRNG1)
```

```
let (int2, intRNG2)
    =  intGenerator.run(intRNG1)
```

```
let (int2, intRNG2)
    =  intGenerator.run(intRNG1)
```

5255

Combinators

```swift
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                     -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```swift
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                                -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```swift
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                                                -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```swift
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                            -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                        -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                    -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```
extension State {
    func map<B>(_ transform: @escaping (A) -> B)
                                 -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (transform(nextA), nextS)
        }
    }
}
```

```swift
let boolGenerator: Rand<Bool>
    = intGenerator.map {int in int % 2 == 1}
```

```
let boolGenerator: Rand<Bool>
    = intGenerator.map {int in int % 2 == 1}
```

```swift
let boolGenerator: Rand<Bool>
    = intGenerator.map {int in int % 2 == 1}
```

```swift
let boolGenerator: Rand<Bool>
    = intGenerator.map {int in int % 2 == 1}
```

```
let boolGenerator: Rand<Bool>
    = intGenerator.map {int in int % 2 == 1}
```

```
let (bool1, boolRNG1)
    =  boolGenerator.run(initialRNG)
```

```
let (bool1, boolRNG1)
    =  boolGenerator.run(initialRNG)
```

false

```
let (bool2, boolRNG2)
    =  boolGenerator.run(boolRNG1)
```

```
let (bool2, boolRNG2)
    =  boolGenerator.run(boolRNG1)
```

```
let (bool2, boolRNG2)
    =  boolGenerator.run(boolRNG1)
```

true

```
let doubleGenerator: Rand<Double>
  = intGenerator.map{int in
                        Double(int)/Double(max)}
```

```
let doubleGenerator: Rand<Double>
  = intGenerator.map{int in
                        Double(int)/Double(max)}
```

```
let doubleGenerator: Rand<Double>
  = intGenerator.map{int in
                         Double(int)/Double(max)}
```

```
let doubleGenerator: Rand<Double>
  = intGenerator.map{int in
                        Double(int)/Double(max)}
```

```
let doubleGenerator: Rand<Double>
  = intGenerator.map{int in
                              Double(int)/Double(max)}
```

```
let (double1, doubleRNG1)
    =  doubleGenerator.run(initialRNG)
```

```
let (double1, doubleRNG1)
     =  doubleGenerator.run(initialRNG)
```

0.5107013500164637

```
let (double2, doubleRNG2)
    =  doubleGenerator.run(doubleRNG1)
```

```
let (double2, doubleRNG2)
    =  doubleGenerator.run(doubleRNG1)
```

```
let (double2, doubleRNG2)
    =  doubleGenerator.run(doubleRNG1)
```

0.8651629897925585

```
struct State<S, A> {
    let run: (S) -> (A, S)
}
```

```
struct IntSet {
    let contains: (Int) -> Bool
}
```

Parser Combinators

```
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}
```

```swift
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}
```

```
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}
```

```
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}
```

```swift
public enum ParserResult<Value> {
    case success(Value, String)
    case failure(String)
}
```

```swift
public enum ParserResult<Value> {
    case success(Value, String)
    case failure(String)
}
```

```
public enum ParserResult<Value> {
    case success(Value, String)
    case failure(String)
}
```

```
public enum ParserResult<Value> {
    case success(Value, String)
    case failure(String)
}
```

```swift
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}
```

```
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}

func run<T>(?)->?{



}
```

```swift
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}

func run<T>(_ parser: Parser<T>,
            on string: String)
                       -> ParserResult<T> {

}
```

```
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}

func run<T>(_ parser: Parser<T>,
            on string: String)
                      -> ParserResult<T> {
    return  parser.parse(string)
}
```

```swift
struct Parser<T> {
    let parse: (String) -> ParserResult<T>
}

func run<T>(_ parser: Parser<T>,
            on string: String)
                     -> ParserResult<T> {
    return  parser.parse(string)
}
```

```
func characterParser(for characterToMatch: Character)
                                              -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                          {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                            String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                  is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                          -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                          {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                             String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                   is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                           -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                          {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                            String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                  is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                           -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                            {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                            String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                  is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                           -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                            {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                            String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                  is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                          -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                         {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                           String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                 is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                                     -> Parser<Character> {
     return Parser<Character>{string in
          guard let firstChar = string.first else
                                   {return .failure("String is empty") }
          if firstChar == characterToMatch {
               return .success(characterToMatch,
                                    String(string.dropFirst()))
          } else { return .failure("\(firstChar) from \(string)
                                           is not \(characterToMatch)")}
     }
}
```

```
func characterParser(for characterToMatch: Character)
                                          -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                            {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                                String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                    is not \(characterToMatch)")}
    }
}
```

```
func characterParser(for characterToMatch: Character)
                                        -> Parser<Character> {
    return Parser<Character>{string in
        guard let firstChar = string.first else
                           {return .failure("String is empty") }
        if firstChar == characterToMatch {
            return .success(characterToMatch,
                            String(string.dropFirst()))
        } else { return .failure("\(firstChar) from \(string)
                                  is not \(characterToMatch)")}
    }
}
```

```
let parseA = characterParser(for: "A")
let parseB = characterParser(for: "B")
```

```
run(parseA, on: "ABC")
```

```
run(parseA, on: "ABC")
```

```
success: A, BC
```

```
run(parseA, on: "ZBC")
```

```
run(parseA, on: "ZBC")
```

failure: Z from ZBC is not A

```
run(parseA, on: "")
```

```
run(parseA, on: "")
```

```
failure: String is empty
```

Combinators

```
let parseAB = parseA.followed(by: parseB)
```

```swift
func followed<U>(by otherParser: Parser<U>)
                                                -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                      -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                        -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                        -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```swift
func followed<U>(by otherParser: Parser<U>)
                                    -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                          -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                    -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
}}}}
```

```swift
func followed<U>(by otherParser: Parser<U>)
                                   -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
}}}}
```

```
func followed<U>(by otherParser: Parser<U>)
                                    -> Parser<(T,U)> {
   return Parser<(T,U)>{string in
       switch self.parse(string) {
       case .failure(let message):
           return .failure(message)
       case .success(let value, let remain):
           switch otherParser.parse(remain) {
           case .failure(let message):
               return .failure(message)
           case .success(let innerValue,
                         let innerRemain):
               return .success((value, innerValue),
                                 innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                    -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
}}}}
```

```
func followed<U>(by otherParser: Parser<U>)
                                             -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                    -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                 innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                        -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
}}}}
```

```
func followed<U>(by otherParser: Parser<U>)
                                        -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
func followed<U>(by otherParser: Parser<U>)
                                    -> Parser<(T,U)> {
    return Parser<(T,U)>{string in
        switch self.parse(string) {
        case .failure(let message):
            return .failure(message)
        case .success(let value, let remain):
            switch otherParser.parse(remain) {
            case .failure(let message):
                return .failure(message)
            case .success(let innerValue,
                          let innerRemain):
                return .success((value, innerValue),
                                innerRemain)
} } } }
```

```
let parseAB = parseA.followed(by: parseB)
```

```
run(parseAB, on: "ABC")
```

```
run(parseAB, on: "ABC")
```

```
success(("A", "B"), "C")
```

```
run(parseAB, on: "ZBC")
```

```
run(parseAB, on: "ZBC")
```

```
failure("Z from ZBC is not A")
```

```
run(parseAB, on: "AZC")
```

```
run(parseAB, on: "AZC")
```

```
failure("Z from ZC is not B")
```

```
run(parseAB, on: "")
```

```
run(parseAB, on: "")
```

```
failure: String is empty
```

```
let parseAorB = parseA.or(parseB)
```

```swift
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                       + message2)
} } } }
```

```swift
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                + message2)
} } } }
```

```
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                + message2)
} } } }
```

```
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                + message2)
} } } }
```

```
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                       + message2)
} } } }
```

```
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                + message2)
} } } }
```

```
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                           + message2)
} } } }
```

```
func or(_ otherParser: Parser) -> Parser {
    return Parser{string in
        switch self.parse(string) {
        case .success(let value, let remain):
            return .success(value, remain)
        case .failure(let message):
            switch otherParser.parse(string) {
            case .success(let value, let remain):
                return .success(value, remain)
            case .failure(let message2):
                return .failure(message + " and "
                                + message2)
} } }}
```

```
let parseAorB = parseA.or(parseB)
```

```
run(parseAorB, on: "ABC")
```

```
run(parseAorB, on: "ABC")
```

```
success: A, BC
```

```
run(parseAorB, on: "ZBC")
```

```
run(parseAorB, on: "ZBC")
```

```
failure: Z from ZBC is not A and
         Z from ZBC is not B
```

```
run(parseAorB, on: "BZD")
```

```
run(parseAorB, on: "BZD")
```

```
success: B, ZD
```

```
run(parseAorB, on: "")
```

```
run(parseAorB, on: "")
```

```
failure: String is empty
```

Combinators …

Henderson's Picture Language

```
struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

```
struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

```
struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

```
struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

```swift
struct PictureFrame {
    let origin: Vector
    let edge1: Vector
    let edge2: Vector
}
```

```
struct PictureFrame {
    let origin: Vector
    let edge1: Vector
    let edge2: Vector
}
```

```
struct Vector {
    let x: CGFloat
    let y: CGFloat
}
```

```swift
struct Vector {
    let x: CGFloat
    let y: CGFloat
}
```

```swift
struct Sketch {
    let paths: [CGPath]
}
```

```
struct Sketch {
    let paths: [CGPath]
}
```

```
public struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

How?

```
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

```
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

```swift
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

```
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

```
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

```
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

```
func pictureFrom(sketch: Sketch) -> Picture {
    return Picture {frame in
        sketch
            .scale(x: frame.edge1.length,
                   y: frame.edge2.length)
            .translate(by: frame.origin)
    }
}
```

And

```swift
public func draw(_ picture: Picture) -> UIView {
    // …
}
```

```swift
public func draw(_ picture: Picture) -> UIView {
  // …
}
```

```
public func draw(_ picture: Picture) -> UIView {
  // …
}
```

```swift
public struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

```swift
public struct Picture {
    let picture: (PictureFrame) -> Sketch
}

extension Picture: CustomPlaygroundDisplayConvertible {



}
```

```
public struct Picture {
    let picture: (PictureFrame) -> Sketch
}

extension Picture: CustomPlaygroundDisplayConvertible {
    public var playgroundDescription: Any {

    }
}
```

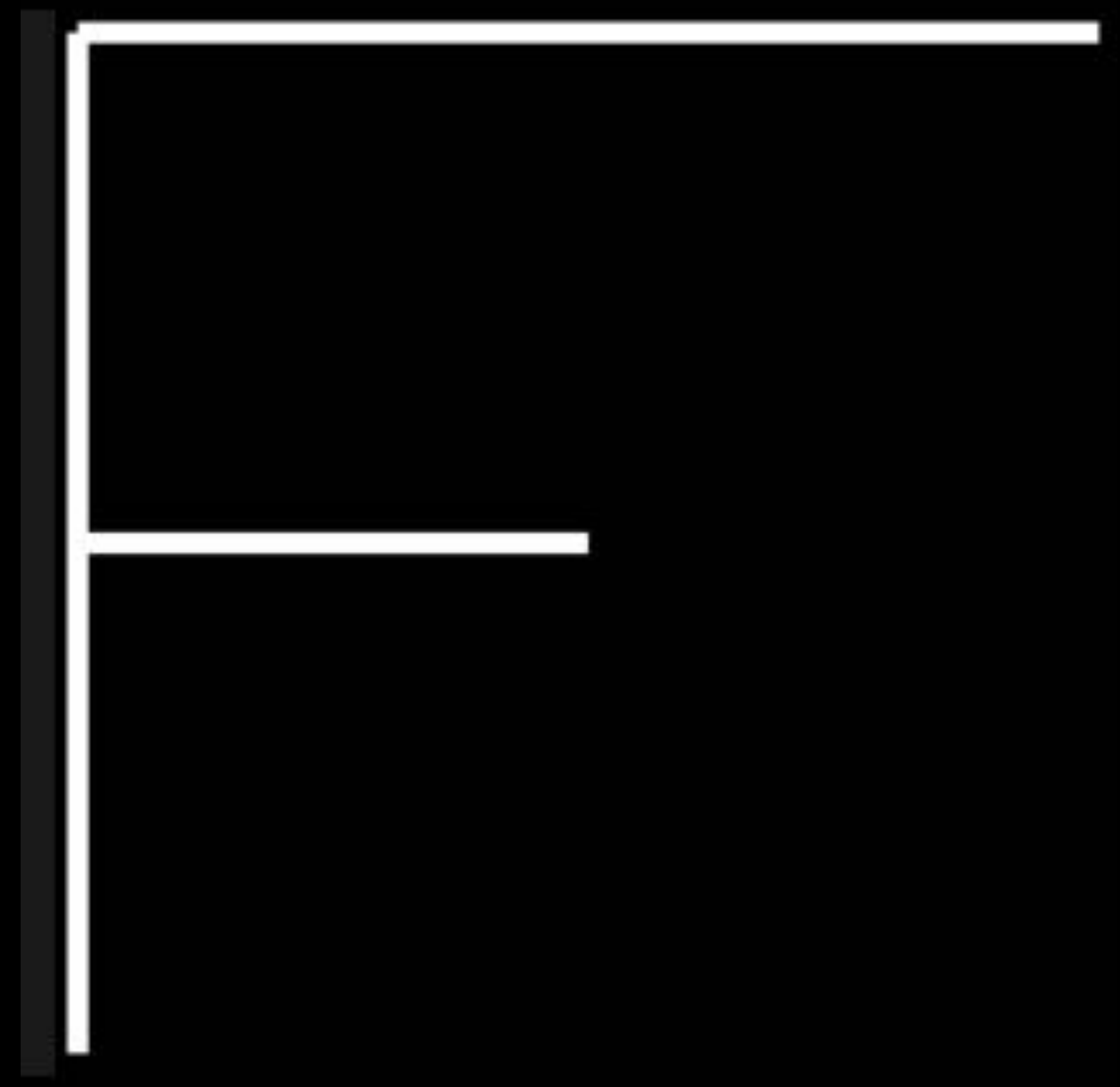
```swift
public struct Picture {
    let picture: (PictureFrame) -> Sketch
}

extension Picture: CustomPlaygroundDisplayConvertible {
    public var playgroundDescription: Any {
        return draw(self)
    }
}
```

draw(f)

~~draw(f)~~

f

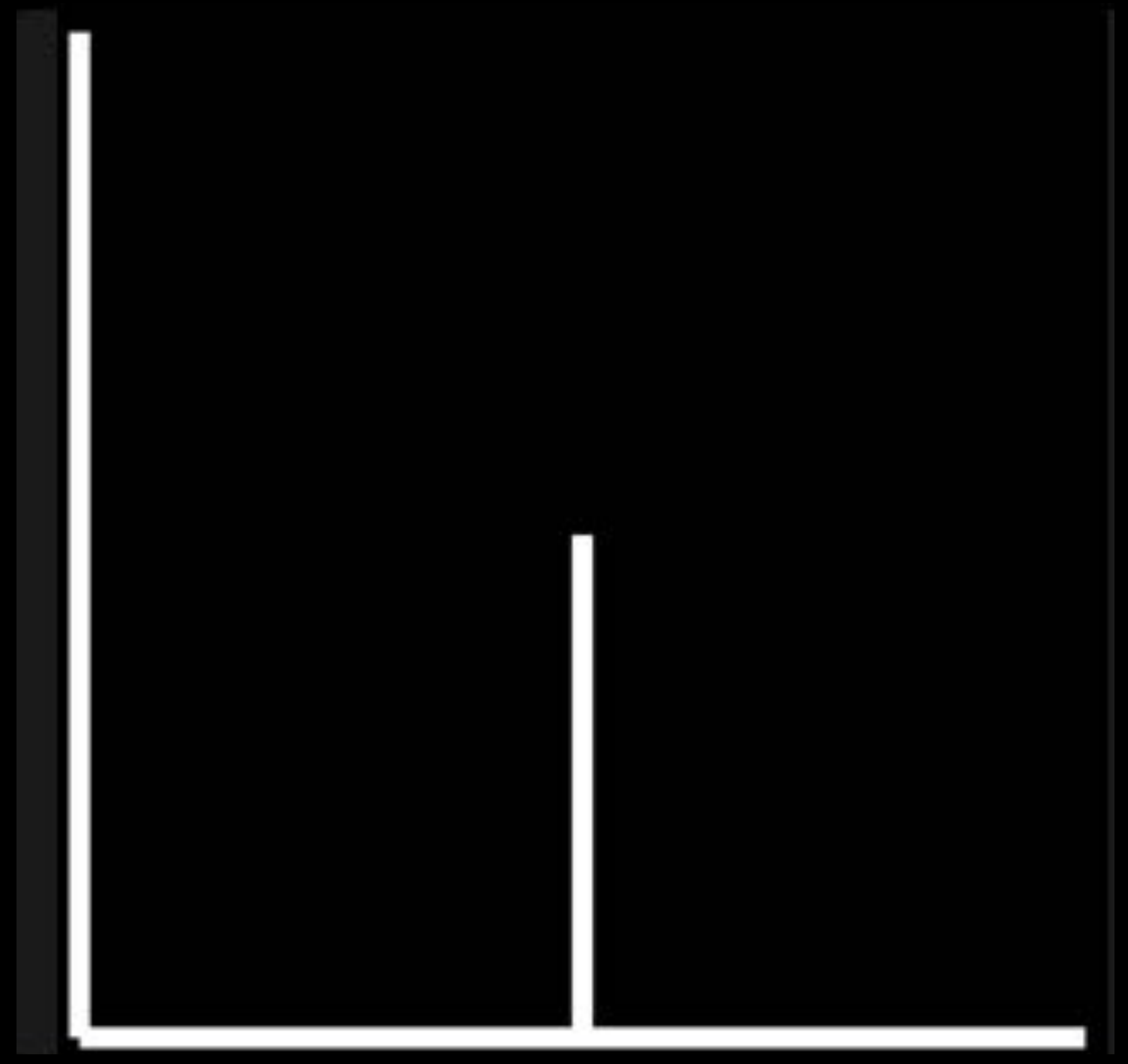
f



Combinators

```
f.rotate()
```

f.rotate()



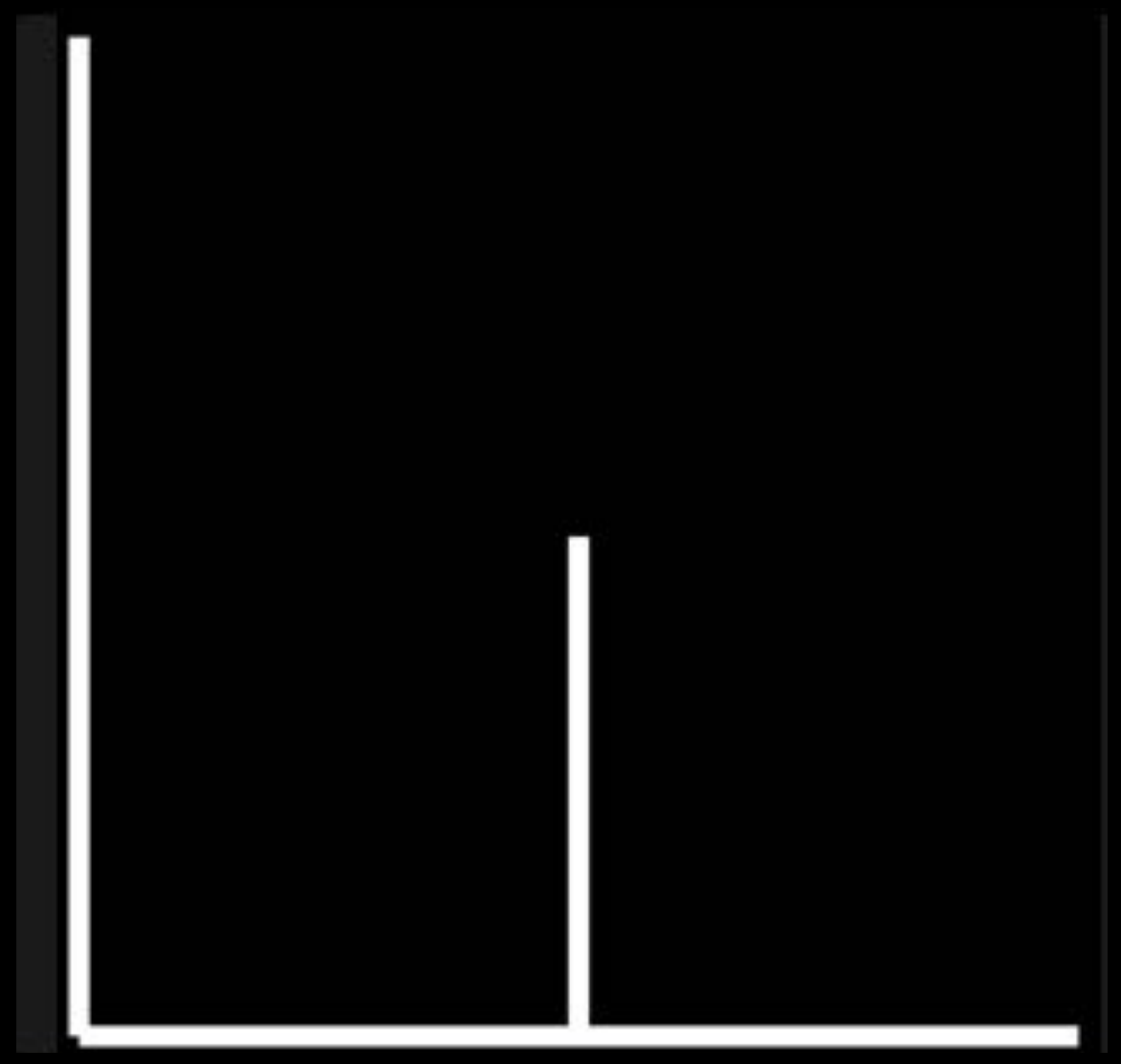
```
func rotate() -> Picture {
    return Picture{frame in
        self.picture(frame)
            .rotateAbout(
                Vector(x: frame.edge1.length/2,
                       y: frame.edge2.length/2),
                by: -CGFloat.pi/2)
    }
}
```

```swift
func rotate() -> Picture {
  return Picture{frame in
      self.picture(frame)
          .rotateAbout(
             Vector(x: frame.edge1.length/2,
                    y: frame.edge2.length/2),
             by: -CGFloat.pi/2)
    }
}
```

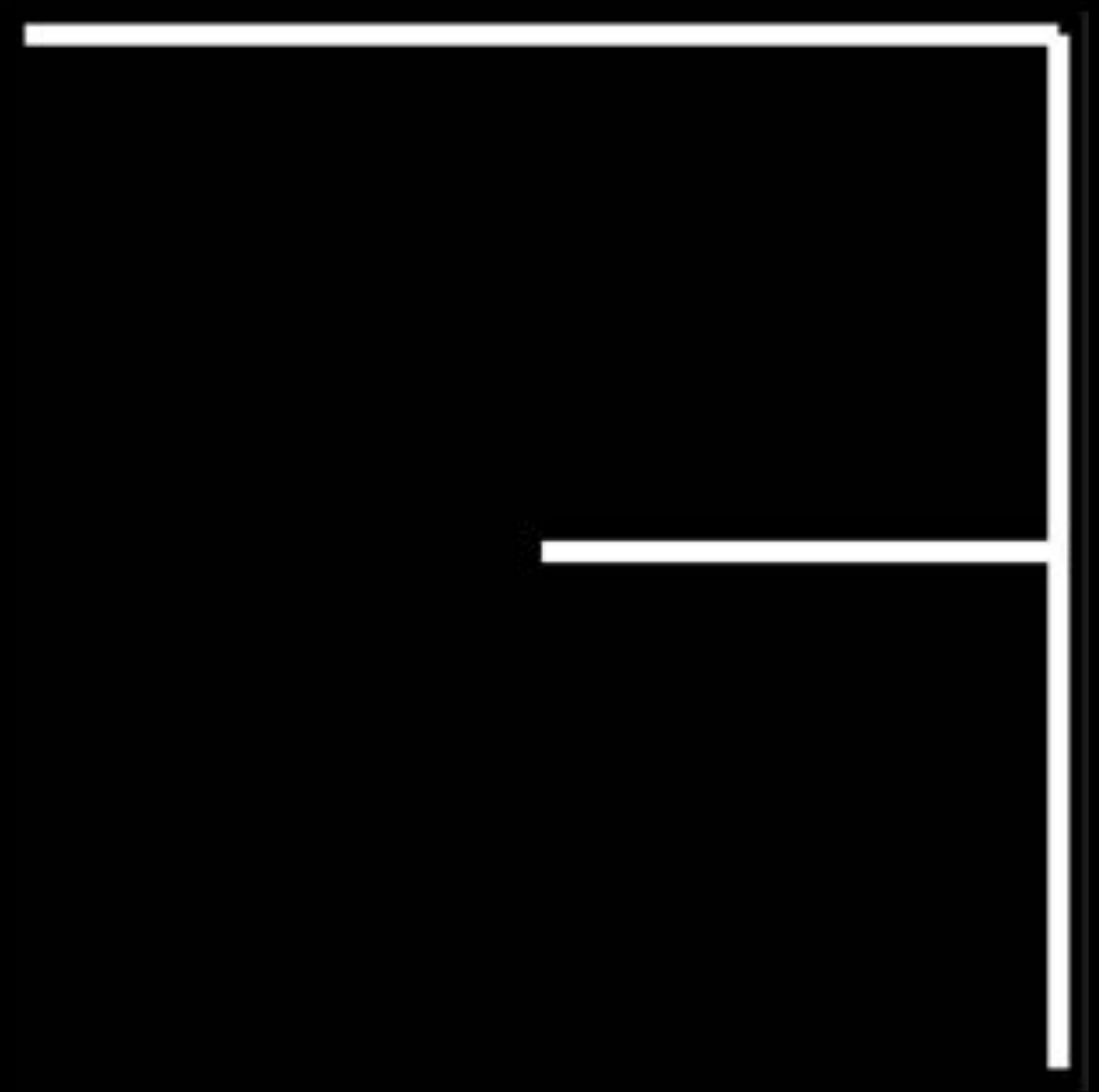
```
func rotate() -> Picture {
  return Picture{frame in
      self.picture(frame)
          .rotateAbout(
              Vector(x: frame.edge1.length/2,
                     y: frame.edge2.length/2),
              by: -CGFloat.pi/2)
     }
}
```

```
func rotate() -> Picture {
   return Picture{frame in
         self.picture(frame)
               .rotateAbout(
                   Vector(x: frame.edge1.length/2,
                             y: frame.edge2.length/2),
                   by: -CGFloat.pi/2)
      }
}
```

f.rotate()



```
f.flipHorizontal()
```

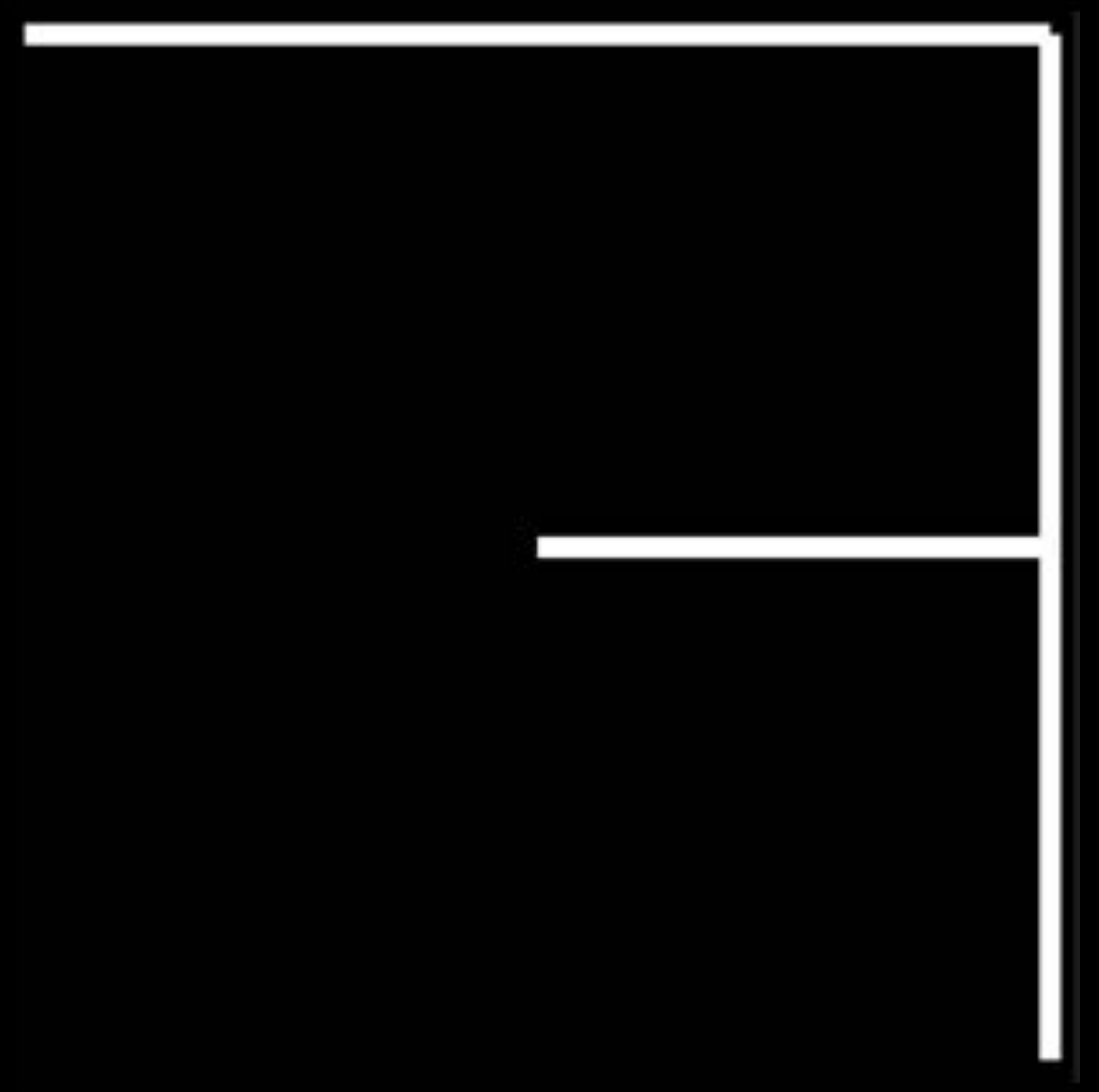


```swift
func flipHorizontal() -> Picture {
    return Picture{frame in
        self.picture(frame)
            .scale(x: -1, y: 1)
            .translate(by:
             Vector(x:frame.edge1.length,
                    y: 0))
    }
}
```

```
func flipHorizontal() -> Picture {
    return Picture{frame in
        self.picture(frame)
            .scale(x: -1, y: 1)
            .translate(by:
             Vector(x:frame.edge1.length,
                    y: 0))
    }
}
```

```
func flipHorizontal() -> Picture {
    return Picture{frame in
        self.picture(frame)
            .scale(x: -1, y: 1)
            .translate(by:
             Vector(x:frame.edge1.length,
                    y: 0))
    }
}
```

f.flipHorizontal()



Combinators

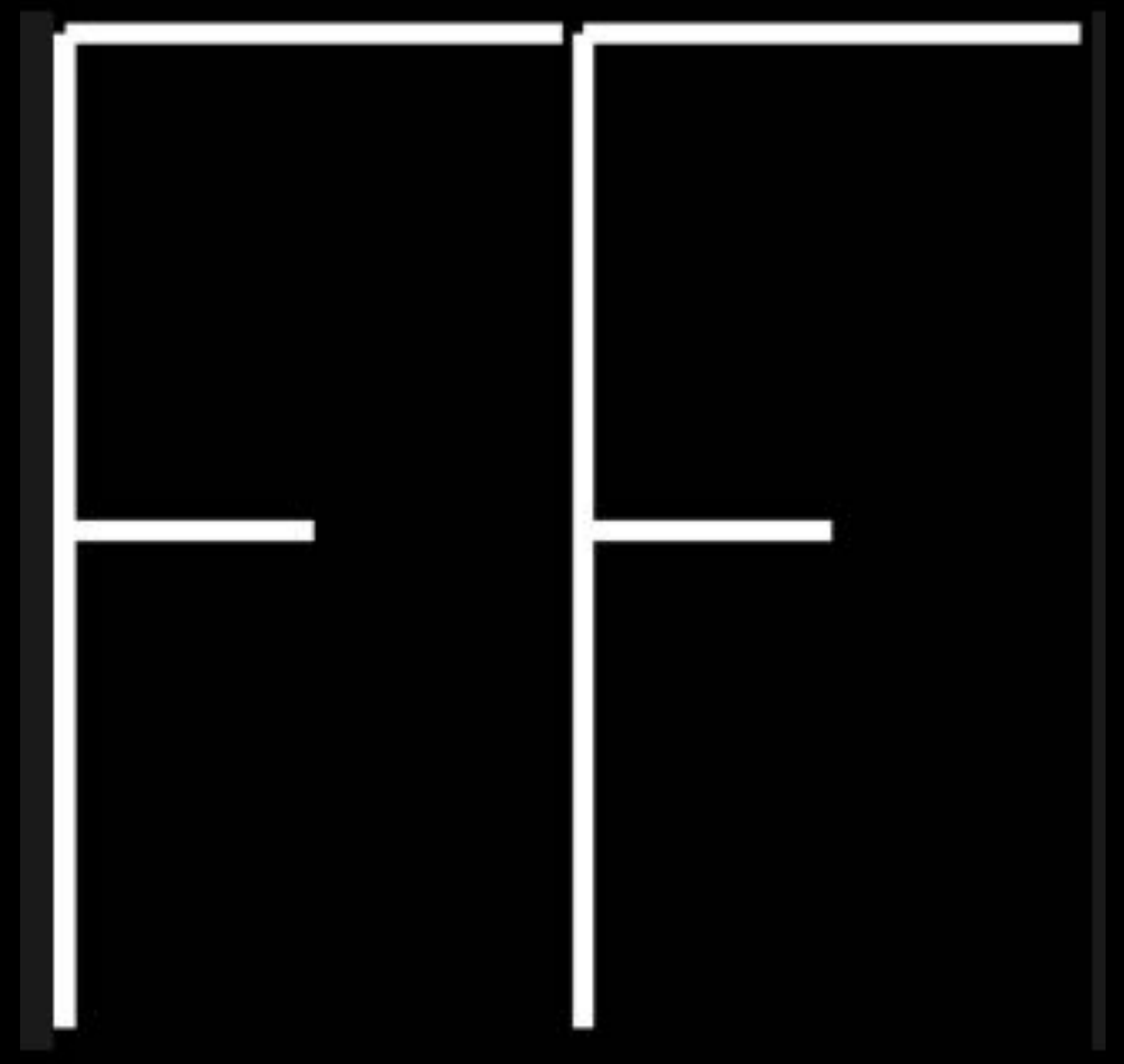
```
(Picture) -> Picture
```

Combinators

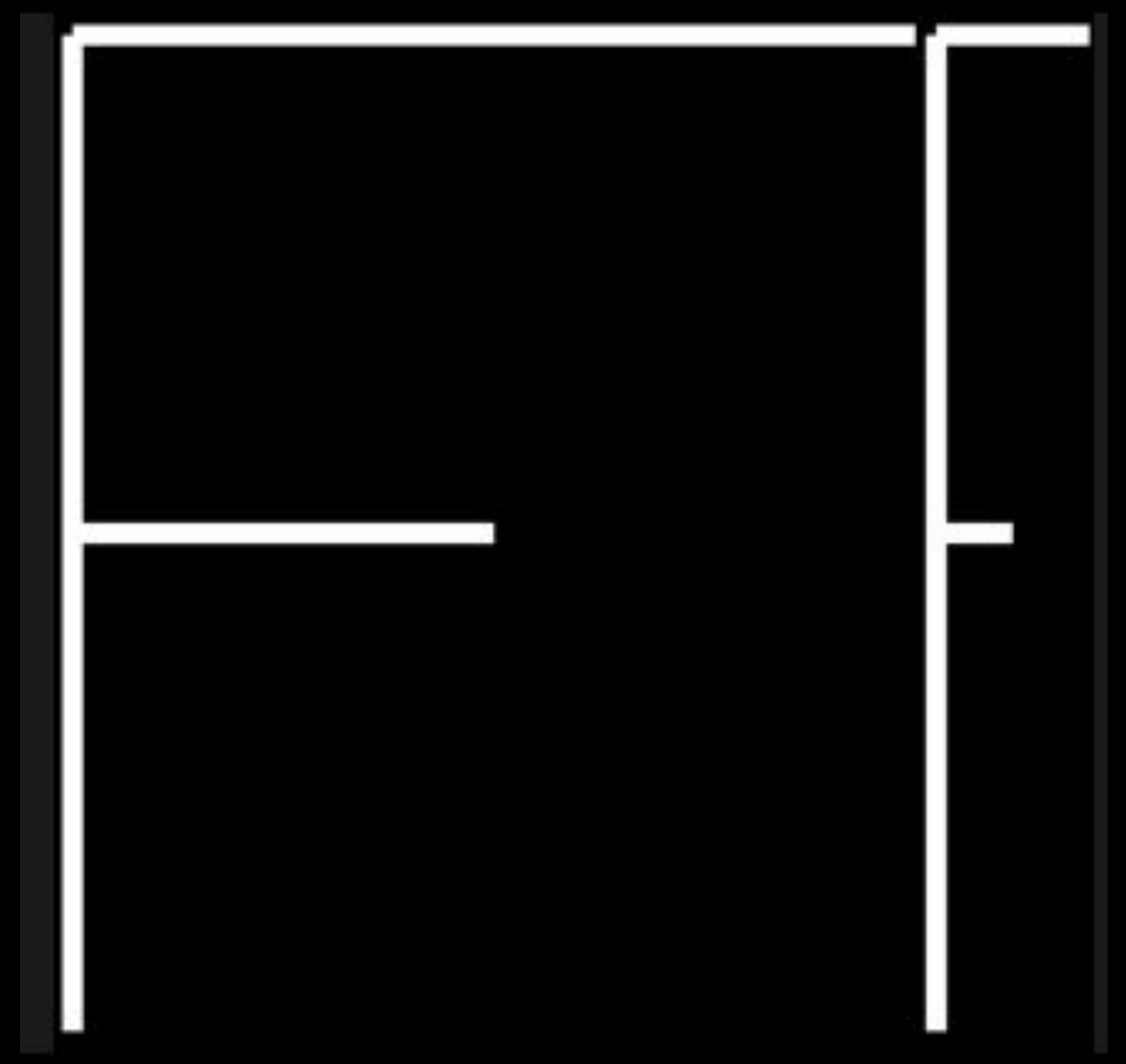
```
(Picture, Picture) -> Picture
```

```
(Picture, Picture…) -> Picture
```

`f.beside(f)`



```
f.beside(f, ratio: 11, to: 2)
```



```swift
func beside(_ otherPicture: Picture,
            ratio leftRatio: Int = 1,
            to rightRatio: Int = 1) -> Picture {
  return Picture {frame in
      let sum = CGFloat(leftRatio + rightRatio)
      return self.picture(frame)
      .scale(x: CGFloat(leftRatio)/sum, y: 1)
      +      otherPicture.picture(frame)
      .scale(x: CGFloat(rightRatio)/sum, y: 1)
      .translate(by: Vector(x: frame.edge1.length
                 * CGFloat(leftRatio)/sum, y: 0))
} }
```

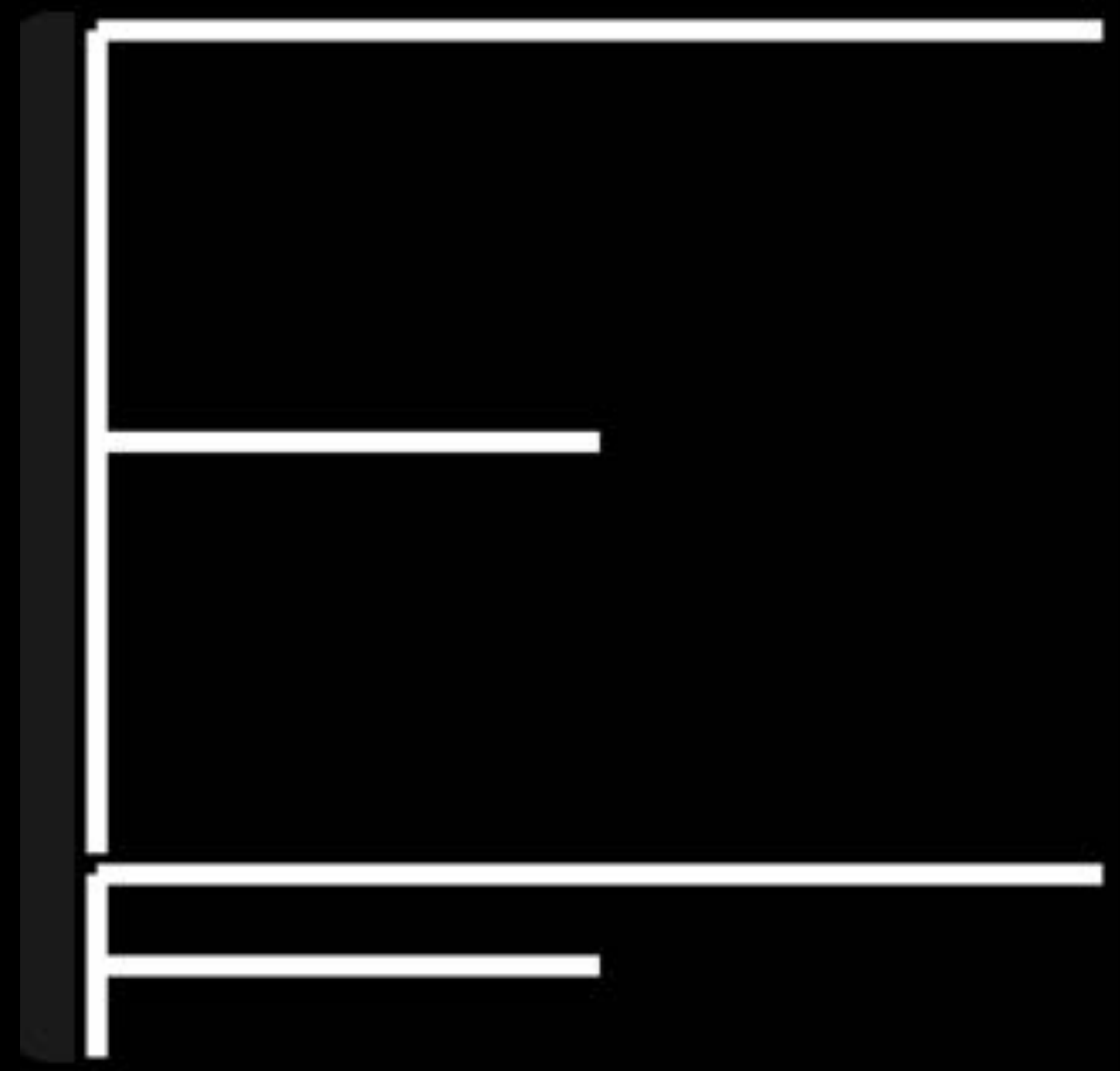
```
func beside(_ otherPicture: Picture,
            ratio leftRatio: Int = 1,
            to rightRatio: Int = 1) -> Picture {
  return Picture {frame in
      let sum = CGFloat(leftRatio + rightRatio)
      return self.picture(frame)
      .scale(x: CGFloat(leftRatio)/sum, y: 1)
      +      otherPicture.picture(frame)
      .scale(x: CGFloat(rightRatio)/sum, y: 1)
      .translate(by: Vector(x: frame.edge1.length
                 * CGFloat(leftRatio)/sum, y: 0))
} }
```

```
func beside(_ otherPicture: Picture,
            ratio leftRatio: Int = 1,
            to rightRatio: Int = 1) -> Picture {
  return Picture {frame in
      let sum = CGFloat(leftRatio + rightRatio)
      return self.picture(frame)
      .scale(x: CGFloat(leftRatio)/sum, y: 1)
      +      otherPicture.picture(frame)
      .scale(x: CGFloat(rightRatio)/sum, y: 1)
      .translate(by: Vector(x: frame.edge1.length
                 * CGFloat(leftRatio)/sum, y: 0))
} }
```

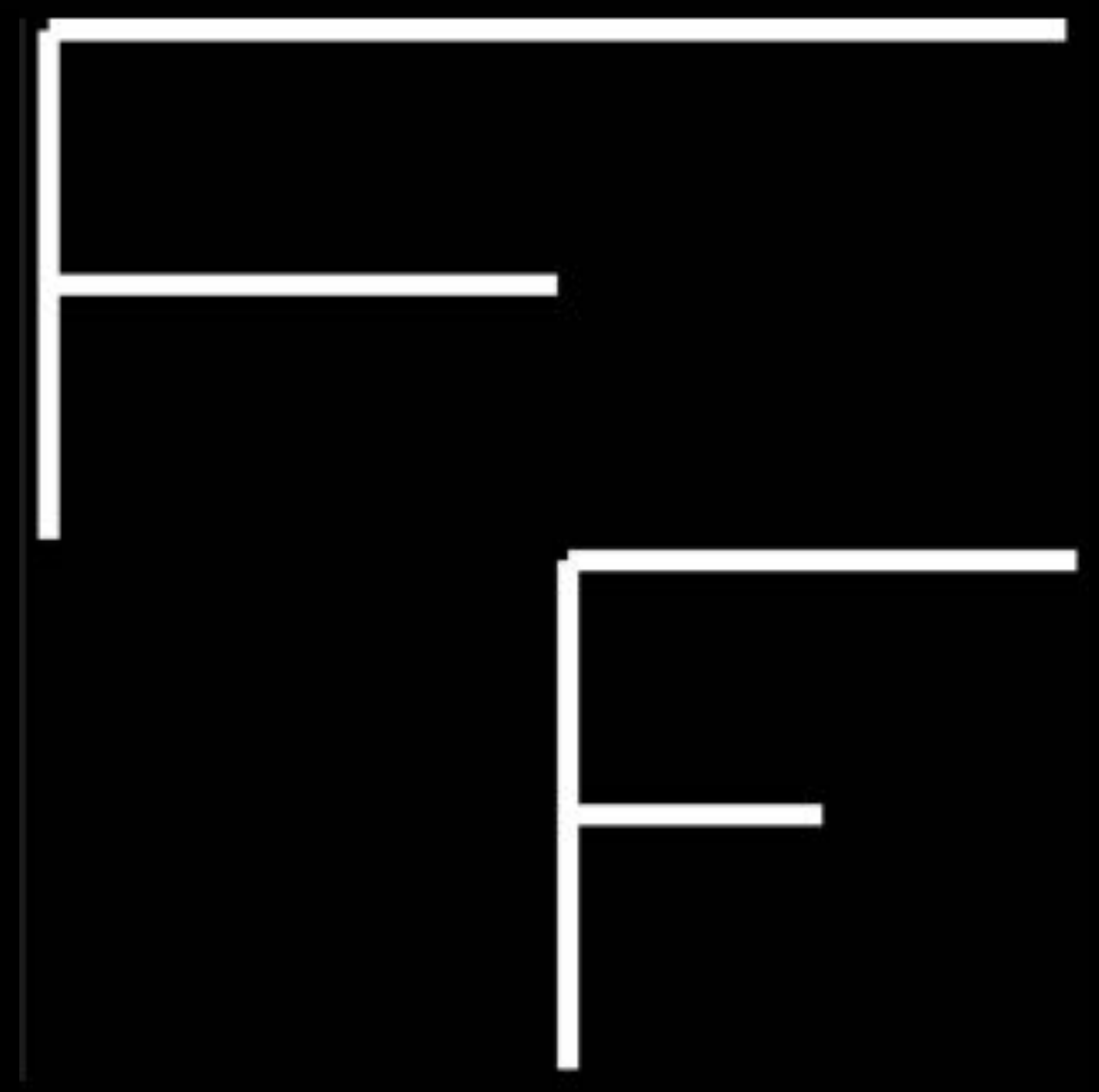
```
func beside(_ otherPicture: Picture,
            ratio leftRatio: Int = 1,
            to rightRatio: Int = 1) -> Picture {
  return Picture {frame in
      let sum = CGFloat(leftRatio + rightRatio)
      return self.picture(frame)
      .scale(x: CGFloat(leftRatio)/sum, y: 1)
      +      otherPicture.picture(frame)
      .scale(x: CGFloat(rightRatio)/sum, y: 1)
      .translate(by: Vector(x: frame.edge1.length
                 * CGFloat(leftRatio)/sum, y: 0))
} }
```

```
f.beside(f, ratio: 11, to: 2)
```

```
f.above(f, ratio: 9, to: 2)
```



```
f.above(blank.beside(f))
```



```swift
func quad(_ b: Picture,
          _ c: Picture,
          _ d: Picture) -> Picture {
    return (self.beside(b))
           .above(c.beside(d))
}
```

```
func quad(_ b: Picture,
          _ c: Picture,
          _ d: Picture) -> Picture {
    return (self.beside(b))
           .above(c.beside(d))
}
```

```
func quad(_ b: Picture,
          _ c: Picture,
          _ d: Picture) -> Picture {
    return (self.beside(b))
           .above(c.beside(d))
}
```

```
func quad(_ b: Picture,
          _ c: Picture,
          _ d: Picture) -> Picture {
    return (self.beside(b))
           .above(c.beside(d))
}
```

```
func quad(_ b: Picture,
          _ c: Picture,
          _ d: Picture) -> Picture {
    return (self.beside(b))
           .above(c.beside(d))
}
```

f.quad()

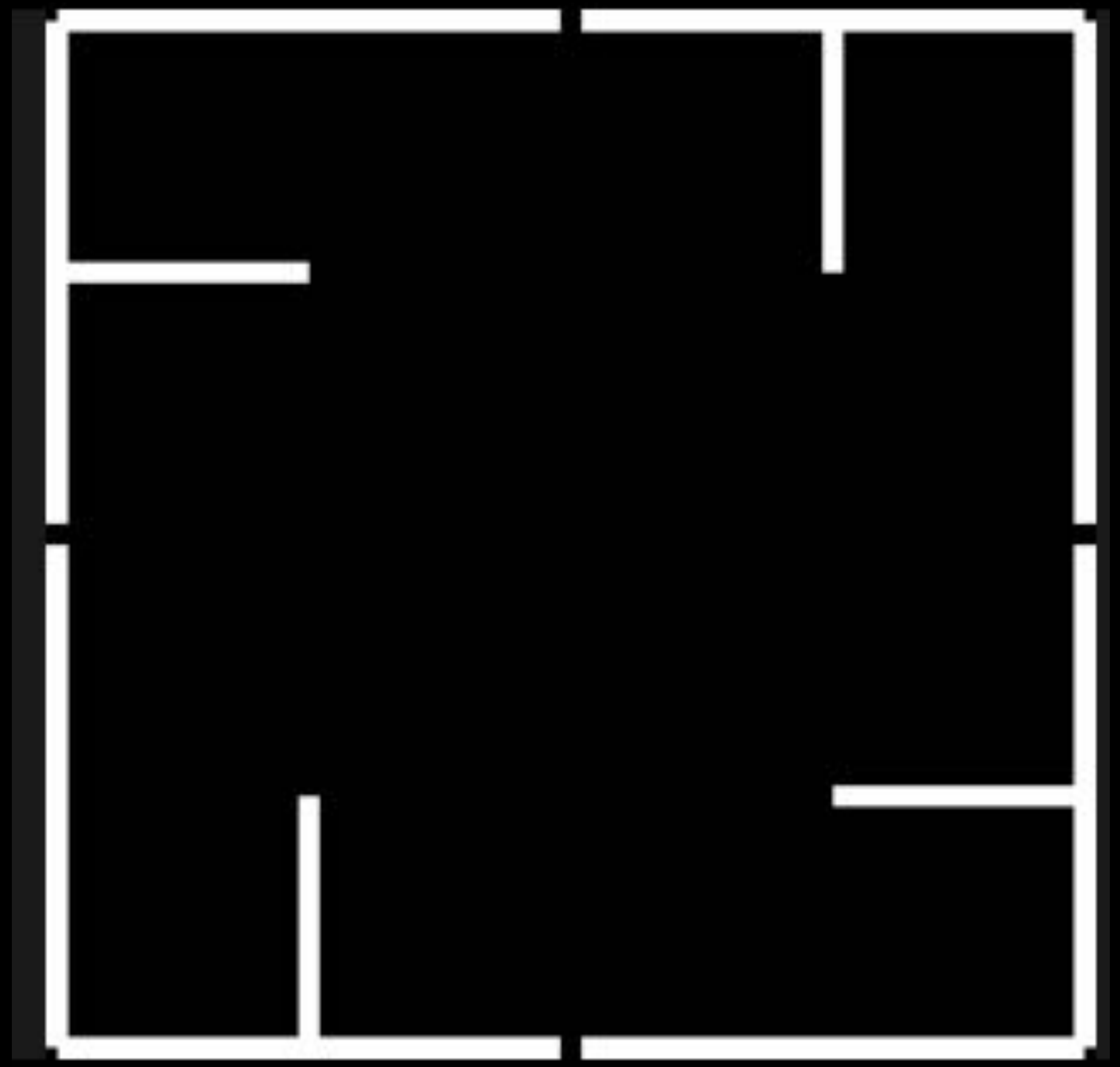
"A combinator is a higher-order function that uses only function application and earlier defined combinators to define a result from its arguments."

https://en.wikipedia.org/wiki/Combinatory_logic (August 18, 2018)

```
func quad(_ b: Picture,
          _ c: Picture,
          _ d: Picture) -> Picture {
    return (self.beside(b))
           .above(c.beside(d))
}
```

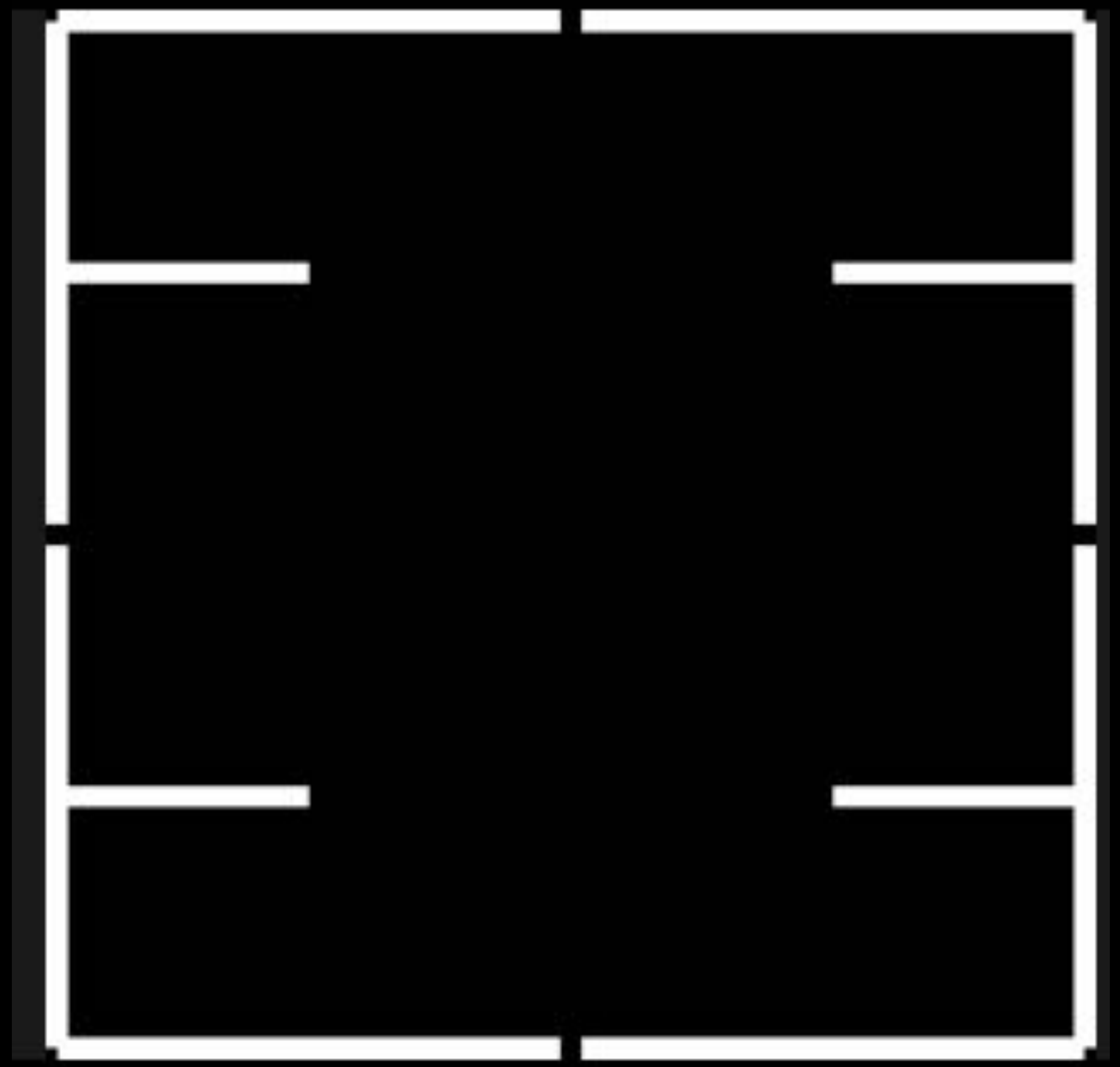
```
func quadRotate(_ b: Picture,
                _ c: Picture,
                _ d: Picture) -> Picture {
    return quad(
             b.rotate().rotate().rotate(),
             c.rotate(),
             d.rotate().rotate())
}
```

f.quadRotate()



```
func quadFlip() -> Picture {
    return quad(flipHorizontal(),
                flipVertical(),
                flipVertical()
                .flipHorizontal())
}
```

f.quadFlip()



```
struct Picture {
    let picture: (PictureFrame) -> Sketch
}
```

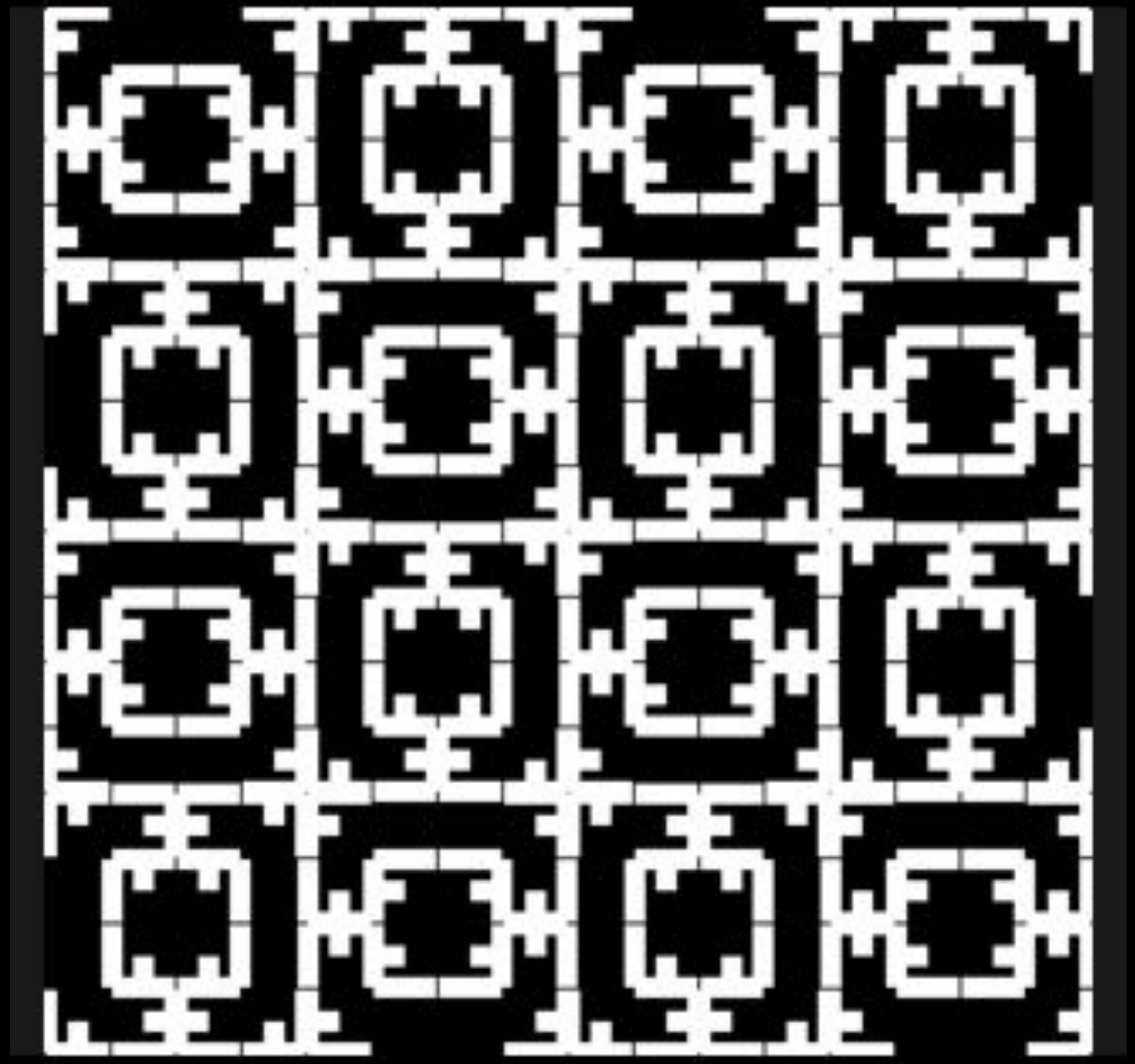
```
f
f.rotate()
f.flipHorizontal()
f.flipVertical()
f.beside(f, ratio: 11, to: 2)
f.above(f.beside(f))
f.above(blank.beside(f))
f.quad()
f.quadRotate()
f.quadFlip()
```

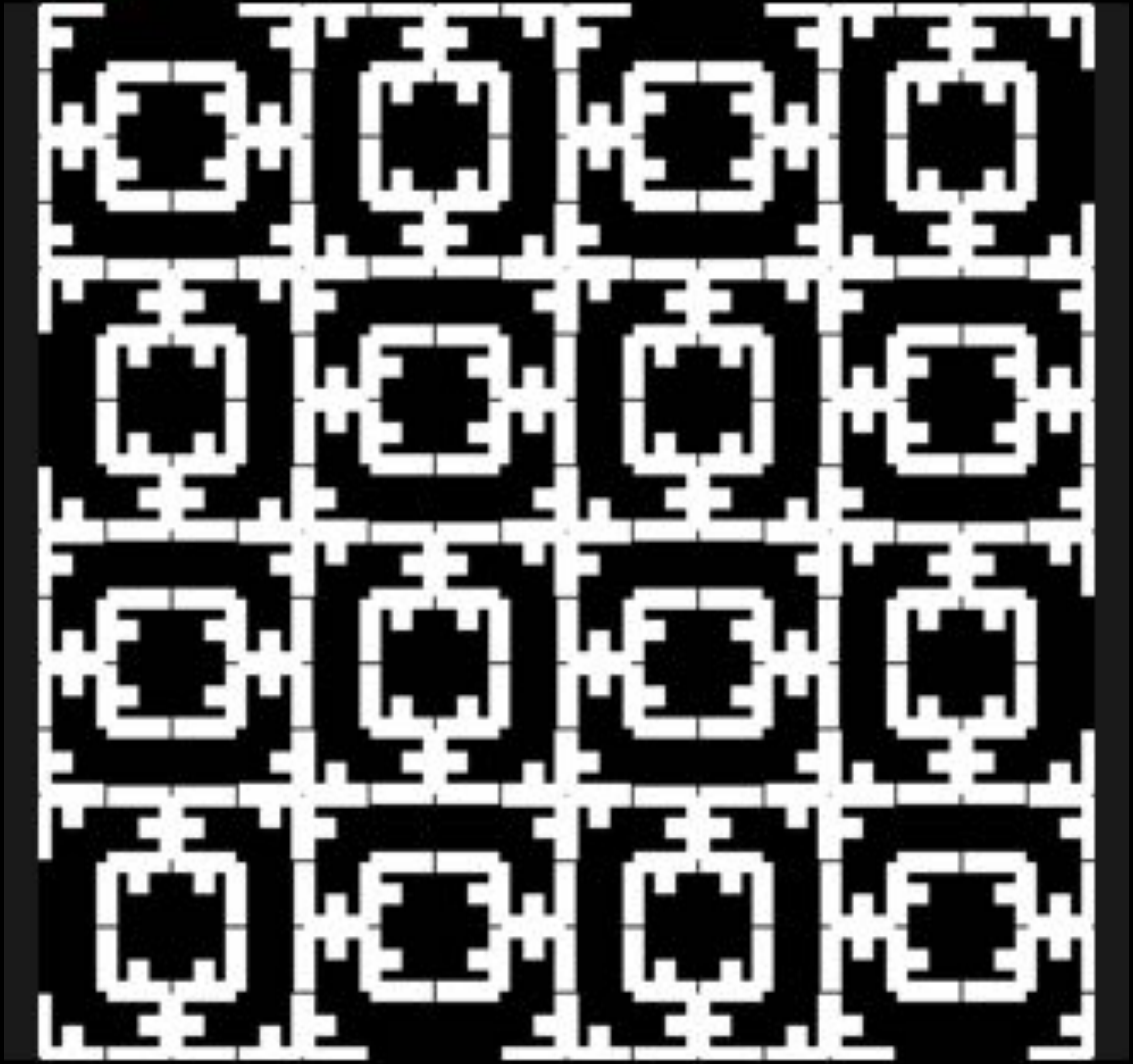
Combinators

```
rotate()
flipHorizontal()
flipVertical()
beside()
above()
quad()
quadRotate()
quadFlip()
```

```
f.quad(blank, f.rotate(), f)
          .quadFlip()
        .quadRotate()
        .quadRotate()
```

```
f.quad(blank, f.rotate(), f)
   .quadFlip()
   .quadRotate()
   .quadRotate()
```





Combinators

NSSpain 2018

Daniel H Steinberg

dimsumthinking.com and editorscut.com