Combinators

Daniel H Steinberg

The Netherlands

Autumn, 2018

http://dimsumthinking.com

Combinators

Daniel H Steinberg

The Netherlands

Autumn, 2018

http://dimsumthinking.com

XCODE 10 : SWIFT 4.2

Combinators in Swift

Swift Training

Recommended Settings

The ePub is best viewed in scrolling mode using the original fonts. The ePub and Mobi versions of this book are best read in single column view.

Contact

Instructor: Daniel H Steinberg

Dim Sum Thinking at http://dimsumthinking.com

View the complete Course List.

email: inquiries@dimsumthinking.com.

Copyright

Legal

TABLE OF CONTENTS

Picture Language

Warmup

Favorites

List

Set

State

Parser Combinator

SECTION 1

Picture Language

The letter F

Add this to the playground page.

```
f
```

That should render the letter F.

Rotate

Add this to the playground.

```
f

f.rotate()

f.rotate().rotate()
f.rotate().rotate().rotate()

f.rotate().rotate().rotate().rotate()
```

Flipping

We can flip a picture horizontally or vertically. We then have to position it back where it was.

Flip horizontally and vertically.

```
f.flipHorizontal()
```

```
f.flipVertical()
```

```
f.flipVertical().flipHorizontal()
```

```
f.flipHorizontal().flipVertical()
```

```
f.rotate().flipHorizontal()
f.flipHorizontal().rotate()
```

Above and Beside

We can combine two pictures into one in which the items are next to or above each other.

```
f.beside(f)
```

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

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

```
f.above(f)
```

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

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

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

Quads

Let's add some hybrids.

```
f.quad()
```

```
f.quad(blank, blank, f)
```

```
f.quadRotate()
```

```
f.quadFlip()
```

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

SECTION 2

Warmup

Setup

We begin with the following in our playground page.

```
import Foundation

let s1 = "Annabelle, my dog, is not the smartest animal in Ohio."
let s2 = "Madam, in Eden, I'm Adam."
```

Array of Words

Let's split `s1` into an array of words by splitting on an empty space.

```
s1.split(separator: " ")
```

```
["Annabelle,", "my", "dog,", "is", "not",
"the", "smartest", "animal", "in", "Ohio."]
```

Now that we have an array, we can sort the elements.

```
s1.split(separator:" ").sorted()
```

Here are the results.

```
["Annabelle,", "Ohio.", "animal", "dog,",
"in", "is", "my", "not", "smartest", "the"]
```

There are two things we'd like to change about the result: (1) The uppercase words should appear alphabetically mixed (case insensitive) with the other words, and (2) We should get rid of the punctuation that follows Annabelle, Ohio, and dog.

Lowercase

The easiest way to sort the words alphabetically without regard to case is to convert the string before turning it into an array.

```
s1.lowercased().split(separator:" ").sorted()
```

The order is correct but this doesn't really give us the results we want as Annabelle and Ohio are now lower case.

```
["animal", "annabelle,", "dog,", "in", "is",
"my", "not", "ohio.", "smartest", "the"]
```

We'll address this in a moment but how might we lower case the words in the list after we split them into an array.

```
s1.split(separator:" ")What goes here.sorted()
```

We can `map` each element of the array like this:

```
s1
    .split(separator:" ")
    .map{word in word.lowercased()}
    .sorted()
```

this

```
s1
    .split(separator:" ")
    .map{$0.lowercased()}
    .sorted()
```

or, after introducing this helper function

```
func lowercased(_ string: Substring) -> String {
    return string.lowercased()
}
```

this point-free approach

```
s1
    .split(separator:" ")
    .map(lowercased)
    .sorted()
```

These results are the same as when we first lowercased the entire string - not quite what we want.

Sorted

Sometimes we forget that there are more higher-order functions than `map`, `filter`, and `reduce`. For example, `sorted` can take an argument that compares two elements of the array and returns `true` if they are in the correct order. If you don't specify this argument then `sorted` uses the default value of `<`.

This is one case where I tend to use `$0` and `$1`.

```
s1.split(separator: " ")
    .sorted{$0.lowercased() < $1.lowercased()}
```

If you'd prefer, you can name the variables.

```
s1.split(separator:" ")
    .sorted{(string1, string2) in
        string1.lowercased() < string2.lowercased()
    }
```

As before, if you prefer a point free approach, create this function

```
func caseInsensitive(_ string1: Substring,
                       string2: Substring) -> Bool {
    return string1.lowercased() < string2.lowercased()
}
```

and use it like this.

```
s1
    .split(separator:" ")
    .sorted(by: caseInsensitive)
```

OK. We've taken care of the sorting. Now let's get rid of those punctuation marks.

Filter

We use `filter` and treat our string like a collection of characters.

There are many ways to do this. Let's filter out all of the characters in the punctuation CharacterSet. We need to convert the characters into unicodescalars and back.

```
s1
```

```
   .filter{(_ char: Character) -> Bool in
        guard let unicodeScalar
            = char.unicodeScalars.first else {return false}
        return !CharacterSet.punctuationCharacters
               .contains(unicodeScalar)
    }
    .split(separator:" ")
    .sorted(by: caseInsensitive)
```

By now you know I prefer to create a helper function.

```
func outPunctuation(_ char: Character) -> Bool {
    guard let unicodeScalar = char.unicodeScalars.first else {return false}
    return !CharacterSet.punctuationCharacters.contains(unicodeScalar)
}
```

I don't know how I feel about this name, but it looks nice in our point free version of our code.

```
s1
    .filter(outPunctuation)
    .split(separator:" ")
    .sorted(by: caseInsensitive)
```

We finally have the results we want.

```
["animal", "Annabelle", "dog", "in", "is",
"my", "not", "Ohio", "smartest", "the"]
```

A Palindrome

For kicks, let's write a routine to see if a string is a palindrome. We usually remove all punctuation and whitespace and convert to lower case before checking that the resulting letters are the same backwards and forwards.

Start with this string

```
let s2 = "Madam, in Eden, I'm Adam."
```

Add this convenience method for removing whitespace - it's almost identical to the punctuation one. You may want to refactor later.

```
func outWhitespace(_ char: Character) -> Bool {
    guard let unicodeScalar = char.unicodeScalars.first else {return false}
    return !CharacterSet.whitespacesAndNewlines.contains(unicodeScalar)
}
```

Now normalize our string.

```
let s3 = s2
           .filter(outPunctuation)
           .filter(outWhitespace)
           .lowercased()
```

This gives us

```
"madaminedenimadam"
```

Now we compare the string with the reversal of the string.

We use `zip` to create a collection of tuples of the elements we're comparing. We compare them with `map`. We reduce to a Bool that tells us whether or not we have a palindrome by anding the values together.

Zip

If we call `reversed()` on a string, we don't get something new so much as we get a new way to traverse our string. So

```
zip(s3, s3.reversed())
```

Gives us

```
Zip2Sequence<String, ReversedCollection<String>>
```

To see the actual pairs, use map.

```
zip(s3, s3.reversed())
   .map{$0}
```

```
[(.0 "m", .1 "m"), (.0 "a", .1 "a"), (.0 "d", .1 "d"), (.0 "a", .1 "a"),
(.0 "m", .1 "m"), (.0 "i", .1 "i"), (.0 "n", .1 "n"), (.0 "e", .1 "e"),
(.0 "d", .1 "d"), (.0 "e", .1 "e"), (.0 "n", .1 "n"), (.0 "i", .1 "i"),
(.0 "m", .1 "m"), (.0 "a", .1 "a"), (.0 "d", .1 "d"), (.0 "a", .1 "a"), (.0
"m", .1 "m")]
```

Replace `s1` for `s2` if you want to see the results for a non-palindrome.

Map

We want to convert this collection of pairs into an array of Bools that is `true` if the elements of the pair is the same and `false` if not.

This is a job for `map`.

Again you can use it with `$0` and `$1`

```
zip(s3, s3.reversed())
    .map{$0 == $1}
```

With variable names

```
zip(s3, s3.reversed())
    .map{(first, second) in first == second}
```

or with the operator `==`.

```
zip(s3, s3.reversed())
    .map(==)
```

```
[true, true, true, true, true, true, true, true,
true, true, true, true, true, true, true, true, true]
```

Reduce

Finally, we can and the results together using reduce to make sure we have nothing but `true`.

```
zip(s3, s3.reversed())
    .map(==)
    .reduce(true){$0 && $1}
```

or with named parameters

```
zip(s3, s3.reversed())
    .map(==)
    .reduce(true){(accumulator, element) in accumulator && element}
```

We can also combine the map and reduce into the reduce

```
zip(s3, s3.reversed())
    .reduce(true){(result, tuple) in
        let (first, second) = tuple
        return result && (first == second)
}
```

`s2` is a palindrome. Check that `s1` isn't.

Notice that we've worked with immutable value types in this section. This is an important mind set.

Map

Sometimes we can use map for Optionals in place of an if let or guard let.

For example, suppose we have this contrived example.

```
func remainder(of string: String) -> Substring? {
    if let firstSpace = string.index(of: " ") {
        return string[firstSpace...]
    } else {
        return nil
    }
}
```

Use it like this.

```
remainder(of: s1)
remainder(of: s3)
```

```
" my dog, is not the smartest animal in Ohio."
nil
```

Here's how we can use `map` to shorten `remainder`.

```
func remainder(of string: String) -> Substring? {
    return string.index(of: " ").map{string[$0...]}
}
```

Map isn't enough

Here's one more example. Let's create a number formatter that spells out numbers and a function that does something fun.

```
let spellOutFormatter = NumberFormatter()
spellOutFormatter.numberStyle = .spellOut
```

```
func next(_ number: Int) -> Int? {
    return spellOutFormatter.string(for: number)?.count
}
```

Create a function that starts with any integer and calls `next` until a stable point is found. In English there is always a stable point.

```
func stableFrom(_ int: Int) -> Int? {
    guard let nextInt = next(int),
        nextInt != int else  {
            return int
    }
    return stableFrom(nextInt)
}
```

Call it

```
stableFrom(1000)
```

```
4
```

`map` almost does what we want.

```
func stableFrom(_ int: Int) -> Int? {
    return next(int).map{$0 == int ? $0 : stableFrom($0) }
}
```

Maybe this is more clear if we're a bit more verbose.

```
func stableFrom(_ int: Int) -> Int? {
    return next(int).map{nextInt in
        if nextInt == int {
            return nextInt
        }
        else {
            return stableFrom(nextInt)
        }
    }
}
```

The function inside of `map` should be from `Int -> Int` but in the `else` branch it's from `Int -> Int?`.

This is exactly what `flatMap` is for - so we don't double wrap optionals.

```
func stableFrom(_ int: Int) -> Int? {
    return next(int).flatMap{nextInt in
        if nextInt == int {
            return nextInt
        }
        else {
            return stableFrom(nextInt)
        }
    }
}
```

Technically we should return `.some(nextInt)` from the `if` clause but Swift (confusingly) does this for us.

We can write `stableFrom` compactly as

```
func stableFrom(_ int: Int) -> Int? {
    return next(int).flatMap{$0 == int ? $0 : stableFrom($0) }
}
```

SECTION 3

Favorites

Setup

The playground contains this code.

```
let words = ["a", "cat", "is", "very", "smart"]
let empty = [String]()

func numberOfLetters(_ string: String) -> Int {
    return string.count
}

func charactersIn(_ string: String) -> [Character] {
    return Array(string)
}

func containsA(_ string: String) -> Bool {
    return string.contains("a")
}
```

We're going to implememt some of the higher order functions. I'll do a demo and draw pictures. Here I just present finished code.

Map

```swift
func myMap(_ input: [String],
           transform:(String) -> Int) -> [Int] {
    var result = [Int]()
    for element in input {
        result.append(transform(element))
    }
    return result
}
```

Generic Map for Collections

```swift
func myMap<Input, Output>(_ input: [Input],
                          transform:(Input) -> Output) -> [Output] {
    var result = [Output]()
    for element in input {
        result.append(transform(element))
    }
    return result
}
```

Map, filter, reduce for Sequence

```swift
extension Sequence {
    func myMap<Output>(transform:(Element) -> Output) -> [Output] {
        var result = [Output]()
        for element in self {
            result.append(transform(element))
        }
        return result
    }

    func myFilter(condition: (Element) -> Bool) -> [Element] {
        var result = [Element]()
        for element in self where condition(element) {
            result.append(element)
        }
        return result
    }

    func myReduce<Output>(_ initialValue: Output,
                          transform: (Output, Element) -> Output) -> Output {
        var accumulator = initialValue
        for element in self {
            accumulator = transform(accumulator, element)
        }
        return accumulator
    }

    func myFlatMap<Output>(transform:(Element) -> [Output]) -> [Output] {
        var result = [Output]()
        for element in self {
            result.append(contentsOf: transform(element))
        }
        return result
    }
}
```

Map and FlatMap for Optional

```
extension Optional {
    func myMap<Output>(transform: (Wrapped) -> Output) -> Output? {
        switch self {
        case .none:
            return .none
        case .some(let value):
            return .some(transform(value))
        }
    }
    func myflatMap<Output>(transform: (Wrapped) -> Output?) -> Output? {
        switch self {
        case .none:
            return .none
        case .some(let value):
            return transform(value)
        }
    }
}
```

Compact Map

```
extension Sequence {
    func myFlatMap<Output>(transform:(Element) -> [Output]) -> [Output] {
        var result = [Output]()
        for element in self {
            result.append(contentsOf: transform(element))
        }
        return result
    }

    func myCompactMap<Output>(transform:(Element) -> Output?) -> [Output] {
        var result = [Output]()
        for element in self {
            if let value = transform(element) {
                result.append(value)
            }
        }
        return result
    }
}
```

SECTION 4

List

The Fundamental Shape

Suppose we want to model a single linked list in Swift. How might we do so?

```
struct List<A> {
        ????? // what goes here?
}
```

We could use an array as the backing store - something like this:

```
struct List<A> {
        private var array: [A] // don't do this
}
```

We're going to model it as an enum. This is probably similar to the way you were taught to think of a list.

```
indirect enum List<A> {
    case empty
    case cons(head: A, tail: List)
}
```

Let's create three lists. One will be empty, a second will contain the int 6, and a third will contain the ints 1,2, and 3.

```
let emptyList = List<Int>.empty
let six = List.cons(head: 6, tail: emptyList)
let oneTwoThree
    = List.cons(head: 1,
                tail: List.cons(head: 2,
                                tail: List.cons(head: 3,
                                                tail: emptyList)))
```

The description is hard to read. Let's add a `description`.

```
extension List: CustomStringConvertible  {
    var description: String {
        switch self {
        case .empty:
            return "( )"
        case let .cons(head, tail):
            return "(\(head)" + tail.description + ")"
        }
    }
}
```

Now the three lists are represented like this:

```
()
(6( ))
(1(2(3( ))))
```

Constructors

Let's make it easier to create a List using variadic parameters. Because they are converted to an array, we'll also have a private init that uses an array of type `A`.

```
extension List {
    private init(_ xs: [A]) {
        if xs.isEmpty {self = List.empty}
        else {
            var xs = xs
            let first = xs.removeFirst()
            self = List.cons(head: first, tail: List(xs))
        }
    }

    init(_ xs: A...){
        self.init(xs)
    }
}
```

We use it like this.

```
let anotherEmptyList = List<Int>()
let five = List(5)
let sevenEightNine = List(7, 8, 9)
```

Convenience Methods

It would be nice to have methods to access the head and tail if the list isn't empty. We're going to throw a fatal error if the list is empty but you could use optionals or something else. Also, here's a convenience method for appending an element to the front of the list.

```
extension List {
    var head: A {
        guard case .cons(let head, _) = self  else {
            fatalError("The list is empty.")
        }
        return head
    }
    var tail: List {
        guard case .cons(_, let tail) = self else {
            fatalError("The list is empty.")
        }
        return tail
    }

    func append(_ element: A) -> List {
        return List.cons(head: element, tail: self)
    }
}
```

Use it like this.

```
sevenEightNine.head
sevenEightNine.tail
sevenEightNine.append(6)
```

```
7
(8(9()))
(6(7(8(9))))
```

Fold Left

In a List there isn't a single reduce like there is in array. In a list it matters as to whether it's a left fold or a right fold. Let's start with fold left as it's easier to implement.

```
extension List {
    func foldLeft<B>(_ initialValue: B, _ f: @escaping(B, A) -> B) -> B {
        guard case let .cons(head, tail) = self else {return initialValue}
        return tail.foldLeft(f(initialValue, head), f)
    }
}
```

Using it is interesting. Here we add.

```
sevenEightNine.foldLeft(0, +)
```

```
24
```

Of course, if you prefer we can use the non-point-free version.

```
sevenEightNine.foldLeft(0){(accumulator, element) in
    accumulator + element
}
```

Now let's add 1 to each element of the list.

```
sevenEightNine.foldLeft(emptyList){(accumulator, element)
    in accumulator.append(element + 1)
}
```

```
(10(9(8( ))))
```

Because of the direction of the fold, we end up reversing the list as we operate on it.

While we're at it, let's add this free functionality in a function we call `reversed`.

```
extension List {
    func foldLeft<B>(_ initialValue: B, _ f: @escaping(B, A) -> B) -> B {
        guard case let .cons(head, tail) = self else {return initialValue}
        return tail.foldLeft(f(initialValue, head), f)
    }
    func reversed() -> List {
        return foldLeft(List.empty){ (reversedList, element) in
            List.cons(head: element, tail: reversedList)
        }
    }
}
```

Use `reversed`.

```
sevenEightNine.reversed()
```

```
(9(8(7())))
```

Fold Right

In Fold right we fold in the other direction. The raw implementation of the method looks like this.

```
extension List {
    func foldLeft<B>(_ initialValue: B, _ f: @escaping(B, A) -> B) -> B {
        guard case let .cons(head, tail) = self else {return initialValue}
        return tail.foldLeft(f(initialValue, head), f)
    }

    func reversed() -> List {
        return foldLeft(List.empty){ (reversedList, element) in
            List.cons(head: element, tail: reversedList)
        }
    }
    func foldRight<B>(_ initialValue: B, _ f: @escaping(A, B) -> B) -> B {
        guard case let .cons(head, tail) = self else {return initialValue}
        return f(head, tail.foldRight(initialValue, f))
    }
}
```

foldLeft is more efficient and easier to follow. foldRight leaves things in proper order.

Of course addition doesn't care which fold we use.

```
sevenEightNine.foldRight(0, +)
```

```
24
```

When we increment the elements of a list, we see that the order is honored.

```
sevenEightNine.foldRight(emptyList){(element, accumulator)
    in accumulator.append(element + 1)
}
```

```
(8(9(10())))
```

Actually, we can implement foldRight using foldLeft and reversed.

```
extension List {
    func foldLeft<B>(_ initialValue: B, _ f: @escaping(B, A) -> B) -> B {
        guard case let .cons(head, tail) = self else {return initialValue}
        return tail.foldLeft(f(initialValue, head), f)
    }

    func reversed() -> List {
        return foldLeft(List.empty){ (reversedList, element) in
            List.cons(head: element, tail: reversedList)
        }
    }

    func foldRight<B>(_ initialValue: B, _ f: @escaping(A, B) -> B) -> B {
        return reversed().foldLeft(initialValue){(b,a) in f(a,b)}
    }
}
```

Map

Increasing each element in a list by 1 is something we'd probably do in a map instead.

Let's define map.

```
extension List {
    func map<B>(_ f: @escaping (A) -> B) -> List<B> {
        return foldRight(List<B>.empty){(element, mappedList) in
            List<B>.cons(head: f(element), tail: mappedList)}
    }
}
```

Now use map to increase each element by 1.

```
sevenEightNine.map{x in x + 1}
```

Filter

We can also derive filter from foldRight.

```
extension List {
    func map<B>(_ f: @escaping (A) -> B) -> List<B> {
        return foldRight(List<B>.empty){(element, mappedList) in
            List<B>.cons(head: f(element), tail: mappedList)}
    }
    func filter(_ f: @escaping (A) -> Bool) -> List {
        return foldRight(List.empty){(element, filteredList) in
          f(element) ? List.cons(head: element, tail: filteredList) : filteredList}
    }
}
```

```
(8(9(10())))
```

Use it, say, to filter out the even numbers.

```
sevenEightNine
    .filter{x in x % 2 == 1}
```

```
(7(9()))
```

More...

There's so much more we can do... flatMap, pure, append, join, ... I leave it as an exercise.

```
extension List {
    func append(_ otherList: List) -> List {
        return foldRight(otherList){(a,b) in List.cons(head: a, tail: b)}
    }
    func join<B>() -> List<B> where A == List<B> {
        return foldRight(List<B>.empty, {(a,b) in a.append(b)})
    }
    func flatMap<B>(_ f: @escaping (A) -> List<B> ) -> List<B> {
        return map(f).join()
    }
}
```

Test it:

```
let lists = List.cons(head: oneTwoThree,
                      tail: List.cons(head: sevenEightNine,
                                      tail: List.empty))

((1(2(3( ))))((7(8(9( ))))( )))

let listsA = oneTwoThree.append(sevenEightNine)

(1(2(3(7(8(9( )))))))

lists.join()

(1(2(3(7(8(9( )))))))

let flatMapDoubleTwoThreFour
    = oneTwoThree.flatMap{x in List.cons(head: x,
                                          tail: List.cons(head: x,
                                                          tail: List<Int>.empty))}

(1(1(2(2(3(3( )))))))
```

Set

Set up

We start with this code in the playground.

```
let smallOdds: Set = [1, 3, 5, 7, 9]
let smallTriples: Set = [3, 6, 9]

smallOdds.contains(3)
smallOdds.contains(4)

smallOdds.union(smallTriples)
smallOdds.intersection(smallTriples)
smallOdds.subtracting(smallTriples)
smallOdds.symmetricDifference(smallTriples)
```

The Challenge

Suppose you were going to create a Set class that contains Ints - perhaps an infinite number of Ints.

Here's a type that we'll create to do this.

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

The only property that `IntSet` has is a function named `contains`.

Here's how we create a set that represents all even integers.

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

Here's how we use it.

```
evens.contains(40024)
evens.contains(603)
```

Description

Add an extension that shows at least part of the set as the set's description.

```
extension IntSet: CustomStringConvertible {
    var description: String {
        return [Int](-10 ... 10).filter{x in contains(x)}.description
    }
}
```

Here's the representation of `evens`.

```
[-10, -8, -6, -4, -2, 0, 2, 4, 6, 8, 10]
```

Here's the empty set and the universal set.

```
let emptySet = IntSet{_ in false}
let universalSet = IntSet{_ in true}
```

Finite sets

Suppose we'd also like to create finite sets using somewhat friendly notation. For example, here's how we might create a range of ints.

```
let twoThroughSeven = IntSet(withRangeFrom: 2, to: 7)
```

And here's how we might create a set with specified elements.

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

Create the inits that make these work.

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

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

Exercise

Create Union, Intersection, and Complement methods for IntSet.

Test them.

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

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

Here we test them.

```
twoThroughSeven.union(primes)
twoThroughSeven.intersection(primes)
twoThroughSeven.complement
```

Preparing for Combinators

Let's create some private helper methods that we'll use to express union, intersection, and complement in a point free way.

```
extension IntSet {
    static func ||(first: IntSet, second: IntSet ) -> IntSet {
        return IntSet{x in first.contains(x) || second.contains(x)}
    }
    static func &&(first: IntSet, second: IntSet ) -> IntSet {
        return IntSet{x in first.contains(x) && second.contains(x)}
    }

    static prefix func !(set: IntSet) -> IntSet {
        return IntSet{x in !set.contains(x)}
    }
}
```

We can now rewrite union, intersection, and complement.

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

    func intersection(_ otherSet: IntSet) -> IntSet {
        return self && otherSet
    }
    var complement:  IntSet {
        return !self
    }
}
```

The returned values remain the same.

More Operators

Let's add operators for finding the sum and difference of two sets and for adding and removing an element.

```
extension IntSet {
    static func -(first: IntSet, second: IntSet ) -> IntSet {
        return first && !second
    }
    static func +(first: IntSet, second: IntSet ) -> IntSet {
        return first || second
    }
    static func -(set: IntSet, element: Int ) -> IntSet {
        return set - IntSet(element)
    }
    static func +(set: IntSet, element: Int ) -> IntSet {
        return set + IntSet(element)
    }
}
```

We're not going to call these directly - we'll use them for other functions.

More Functionality

Let's implement set difference (minus), symmetric difference, and support for adding and removing an element.

```
extension IntSet {
    func minus(_ setToBeRemoved:  IntSet) -> IntSet {
        return self.intersection(setToBeRemoved.complement)
    }

    func symmetricDifference(with otherSet: IntSet) -> IntSet {
        return self.union(otherSet).minus(self.intersection(otherSet))
    }

    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
        }
    }
}
```

Here we take these methods for a test drive.

```
twoThroughSeven.minus(primes)
twoThroughSeven.symmetricDifference(with: primes)
twoThroughSeven.add(0)
twoThroughSeven.remove(0)
twoThroughSeven.add(0)
twoThroughSeven.remove(0)
```

Looking back and forward

Consider again this code:

```
twoThroughSeven.union(primes)
twoThroughSeven.intersection(primes)
twoThroughSeven.complement

twoThroughSeven.minus(primes)
twoThroughSeven.symmetricDifference(with: primes)
twoThroughSeven.add(0)
twoThroughSeven.remove(0)
twoThroughSeven.add(0)
twoThroughSeven.remove(0)
```

It's striking that the definition of `IntSet` is this:

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

The only state that an `IntSet` contains is a function.

More...

What would it take to make IntSet generic? Can we? Could we implement map and filter on these infinite sets?

SECTION 6

State

A Simple Idea that Isn't

Here was our generic Set.

```
struct MySet<A> {
    let contains: (A) -> Bool
}
```

This is State:

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

Add that definition of State to the playground.

You'll hear this referred to as "the state monad".

An example

Check the sources directory for RNG.swift. This is a sample (not to be shipped) implementation of a random number generator.

```
import Foundation

public let max = 6074

public let initialRNG = RNG(seed: 17)

public struct RNG {
    private let A = 107
    private let C = 1283
    private let M = 6073

    public let seed: Int

    public init(seed: Int) {
        self.seed = seed
    }

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

Note that RNG is a random number generator that produces the next Int along with the next RNG that you'll use to get the int after that.

Look back at the `run` method from `State` and let's treat `S` as `RNG`.

In other words, add this typealias to the playground.

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

Int Generator

Let's start by instantiating and using a RNG for Ints.

```
let randomIntGenerator: Rand<Int> = State(run: {rng in rng.nextInt()})
```

Now use it.

```
let (n1, rng1) = randomIntGenerator.run(initialRNG)
n1

3102
let (n2, rng2) = randomIntGenerator.run(rng1)
n2

5255
```

Couldn't we have just used RNG without State?

Yes - but now we can take advantage of State being generic in A.

Map

Suppose we wanted a random Bool generator. We could write a version of RNG for Bool. OR we can take out Int RNG and a function from Int to Bool and use map to get a random Bool generator.

Here's map

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

Create a Bool RNG with map and a function from Int -> Bool

```
let randomBoolGenerator = randomIntGenerator.map{int in int % 2 == 1}
```

Use it like this.

```
let (b1, brng1) = randomBoolGenerator.run(initialRNG)
b1

false
let (b2, brng2) = randomBoolGenerator.run(brng1)
b2

true
```

Similarly we can create a random number generator for doubles between 0 and 1.

```
let randomDoubleGenerator = randomIntGenerator.map{int in Double(int)/Double(max)}
```

As with the others, we use it like this.

```
max

6074

let (d1, drng1) = randomDoubleGenerator.run(initialRNG)
d1

0.5107013500164637
```

```
let (d2, drng2) = randomDoubleGenerator.run(drng1)
d2

0.8651629897925585
```

Map isn't enough

Suppose we want the next random integer less than a specified number. For example the next random integer less than 100. Our generator has to keep generating the next integer until the condition is met. It looks something like this.

BTW the following doesn't compile.

```
extension State where S == RNG, A == Int {
    func nextLessThan(_ upperLimit: Int) -> State<RNG, Int> {
        return map{int in
            guard int < upperLimit else {
                print(int)
                return self.nextLessThan(upperLimit)
            }
            return int
        }
    }
}
```

The problem is the function we pass into `map` has different return types on each branch. The main branch is fine - it's `Int -> Int`. The `else` branch is a problem as it's `Int -> State<RNG, Int>`.

We saw something similar during our warmup. We need `flatMap`.

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

Now change the `map` in `nextLessThan` to `flatMap`.

```
extension State where S == RNG, A == Int {
    func nextLessThan(_ upperLimit: Int) -> State<RNG, Int> {
        return flatMap{int in
            guard int < upperLimit else {
                print(int)
                return self.nextLessThan(upperLimit)
            }
            return int
        }
    }
}
```

We've fixed the recursive call but broken the main return.

Pure

Again, you might remember that in our warmup, Swift automatically wrapped our optional so we didn't have to wrap it using

`.some`. In general the issue is that in the previous example we needed to return an Optional and here we need to return a State instance.

The `pure` function is used to embed an element in the world of Arrays, Optionals, State, ...

It is either implemented as a static or as an init

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

Now we can fix `nextLessThan`.

```
extension State where S == RNG, A == Int {
    func nextLessThan(_ upperLimit: Int) -> State<RNG, Int> {
        return flatMap{int in
            guard int < upperLimit else {
                print(int)
                return self.nextLessThan(upperLimit)
            }
            return State<RNG, Int>.pure(int)
        }
    }
}
```

Run it

```
let lessThan1000: Rand<Int> = randomIntGenerator.nextLessThan(1000)

let (nextLessThan1000, ltrng1) = lessThan1000.run(initialRNG)

109
```

Map, FlatMap, Pure

You can always derive `map` from `flatMap` and `pure`. This is why every Monad is automatically a Functor

```
extension State {
    static func pure(_ a: A) -> State<S, A> {
        return State<S, A>{s in (a, s)}
    }
    func map<B>(_ f: @escaping (A) -> B) -> State<S, B> {
        return State<S, B>{s in
            let (nextA, nextS) = self.run(s)
            return (f(nextA), nextS)
        }
    }
    func flatMap<B>(_ f: @escaping  (A) -> State<S, B>)
        -> State<S, B> {
            return State<S,B>{s in
                let (nextA, nextState) = self.run(s)
                return f(nextA).run(nextState)
            }
    }
        func derivedMap<B>(_ f: @escaping (A) -> B) -> State<S, B> {
        return flatMap{a in State<S,B>.pure(f(a))}
    }
}
```

One last example

How might we calculate the next n random numbers all at once?

```
extension State {
    func next(_ n: Int, accumulator: [A] = [A]()) -> State<S, [A]> {
        return flatMap{a in
            guard n > 0 else { return State<S, [A]>.pure(accumulator)
            }
            return self.next(n - 1, accumulator: accumulator + [a] )
        }
    }
}
```

```
let next100 = intGenerator.next(100)

let (result100, nextRNG) = next100.run(initialRNG)
```

```
[3102, 5255, 4852, 4242, 5775, 5835, 109, 800, 1861,
1, 1390, 4261, 1735, 4738, 4190, 211, 5641, 3643, ...
4612, 2854, 3011, 1591, 1476, 1317, 2523, 4032, 1524]
```

SECTION 7

Parser Combinator

Set up

Here is the `ParserResult` type in the source directory.

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

For convenience there's also a `description`.

```
extension ParserResult: CustomStringConvertible where Value: CustomStringConvertible
 {
    public var description: String {
        switch self {
        case .failure(let message):
            return "failure: " + message
        case .success(let value, let remaining):
            return "success: \(value), \(remaining)"
        }
    }
}
```

The Parser

Let's define the Parser in the playground page.

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

That's it. A Parser holds a function that takes a String and returns a ParserResult.

It's kind of like Set. It contains a function that takes an input and returns an output.

It's kind of like State. If I successfully parse then my ParseResult contains the resulting String that I can pipe into the next parser.

Sample Parser

So when we create a parser we have to define the function.

```
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 \(characterToMa
tch)")}
    }
}
```

We could have chosen to treat the String being empty as just part of the general failure.

Create and run our parser

Create an instance of the parser that parses the letter "A"

```
let parserA = characterParser(for: "A")
```

Use parserA

```
parserA.parse("ABC")
```

```
success: A, BC
```

```
parserA.parse("ZBC")
```

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

```
parserA.parse("")
```

```
failure: String is empty
```

We're able to parse a single letter.

Followed By

Let's expand our example to parse for one character followed by another.

```
extension Parser {
    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)
                }
            }
        }
    }
}
```

To use it, add a second character parser for "B".

```
let parserA = characterParser(for: "A")

parserA.parse("ABC")
parserA.parse("ZBC")
parserA.parse("")

let parserB = characterParser(for: "B")
let parserAThenB = parserA.followed(by: parserB)

parserAThenB.parse("ABC")

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

parserAThenB.parse("ZBC")

failure("Z from ZBC is not A")

parserAThenB.parse("AZC")

failure("Z from ZC is not B")

parserAThenB.parse("")
```

```
failure("String is empty")
```

Or

Let's implement a simple case of or where the types of what we're parsing for are the same.

```
extension Parser {
    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 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)
                }
            }
        }
    }
}
```

Test it.

```
let parserAOrB = parserA.or(parserB)

parserAOrB.parse("ABC")

success: A, BC

parserAOrB.parse("ZBC")

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

parserAOrB.parse("BZC")

success: B, ZC

parserAOrB.parse("")

failure: String is empty and String is empty
```