objc ↑↓ Functional Programming in Swift

By Chris Eidhof, Florian Kugler and Wouter Swierstra

Contents

Note: This is a pre-release version. Many things are missing.

Thanks for buying the pre-release book. We're very happy that you chose to support us. We're planning to publish the first content early July. We'll be updating this PDF on a regular basis, but for the latest available content, check out https://github.com/objcio/fpinswift-beta

We would love to hear from you. If there are any topics you'd like to see included, or if there's anything that deserves a better explanation, let us know. File an issue on GitHub, or send an email to mail@objc.io

This pdf was generated on Friday 8th August, 2014; the code was tested using Xcode 6 beta 5.

Florian, Wouter and Chris

Introduction

Why write this book? There is plenty of documentation on Swift readily available from Apple, and there are many more books on the way. Why does the world need yet another book on yet another programming language?

This book tries to teach you to think functionally. We believe that Swift has the right language features to teach you how to write functional programs. But what makes a program functional? Or why bother learning about this in the first place? In his paper, "Why Functional Programming Matters," John Hughes writes:

> Functional programming is so called because its fundamental operation is the application of functions to arguments. A main program itself is written as a function that receives the program's input as its argument and delivers the program's output as its result.

So rather than thinking of a program as a sequence of assignments and method calls, functional programmers emphasize that each program can be repeatedly broken into smaller and smaller pieces. By avoiding assignment statements and side effects, Hughes argues that functional programs are more modular than their imperative- or object-oriented counterparts. And modular code is a Very Good Thing.

In our experience, learning to think functionally is not an easy thing. It challenges the way we've been trained to decompose problems. For programmers who are used to writing for-loops, recursion can be confusing; the lack of assignment statements and global state is crippling; and closures, generics, higher-order functions, and monads are just plain weird.

Throughout this book, we will assume that you have previous programming experience in Objective-C (or some other object-oriented language). We won't cover Swift basics or teach you to set up your first XCode project, but we will try to refer to existing Apple documentation when appropriate. You should be comfortable reading Swift programs and familiar with common programming concepts, such as classes, methods, and variables. If you've only just started to learn to program, this may not be the right book for you.

In this book, we want to demystify functional programming and dispel some of the prejudices people may have against it. You don't need to have a PhD in mathematics to use these ideas to improve your code! We don't want to claim that functional programming is the only way to program in Swift. Instead, we believe that learning about functional programming add an important new tool to your toolbox that will make you a better developer in any language.

Chapter 1

Thinking Functionally

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes.

What this chapter is about

Functions in Swift are first-class values, i.e., functions may be passed as arguments to other functions. This idea may seem strange if you're used to working with simple types, such as integers, booleans or structs. In this chapter, we will try to motivate why first-class functions are useful and give a first example of functional programming in action.

Example: Battleship

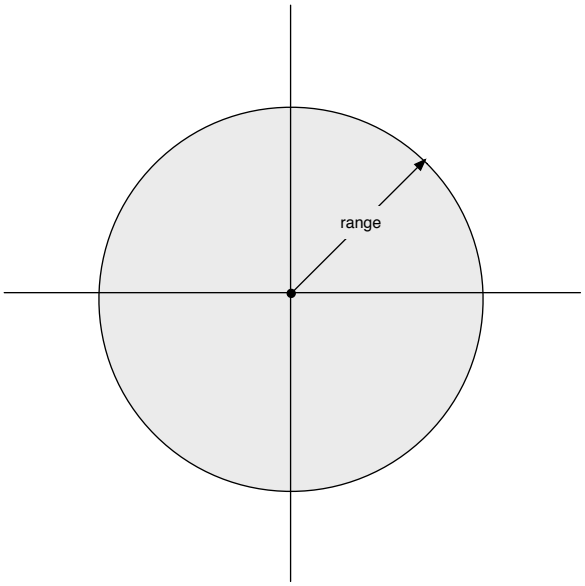
We'll introduce first-class functions using the example of an algorithm you would implement if you would want to create a Battleship-like game.[1] The problem we'll look at boils down to determining whether or not a given point is in range, without being too close to friendly ships or ourselves.

As a first approximation, you might write a very simple function that checks whether or not a point is in range. For the sake of simplicity, we will assume that our ship is located at the origin. We can visualize the region we want to describe as follows:

The first function we write, `inRange1` checks when a point is in the grey area in the picture above. Using some basic geometry, we can write this function as follows:

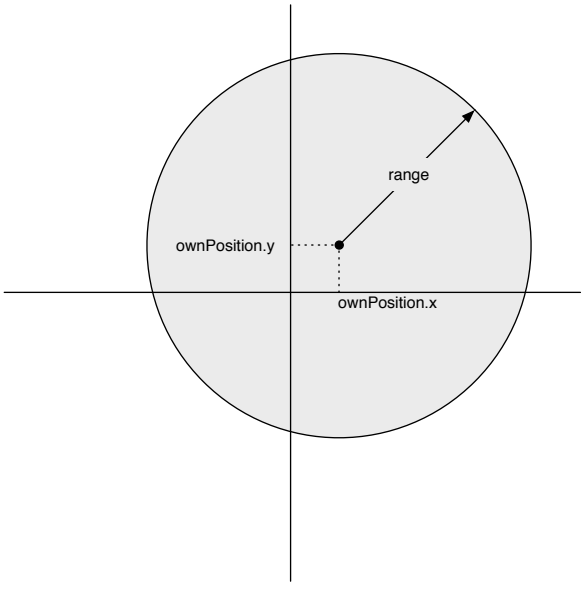
```
typealias Position = CGPoint
typealias Distance = CGFloat

func inRange1(target: Position, range: Distance) -> Bool {
   return sqrt(target.x * target.x + target.y * target.y) <= range
}
```

[1] The code presented here is inspired by the Haskell solution to a problem posed by the ARPA documented here: Hudak, Paul, and Mark P. Jones. Haskell vs. Ada vs. C++ vs. awk vs.... an experiment in software prototyping productivity. Technical report, Yale University, Dept. of CS, New Haven, CT, 1994.



Note that we are using Swift's typealias construct let us introduce a new name for an existing type. From now on, whenever we write `Position`, feel free to read `CGPoint`, a pair of an x and y coordinate.

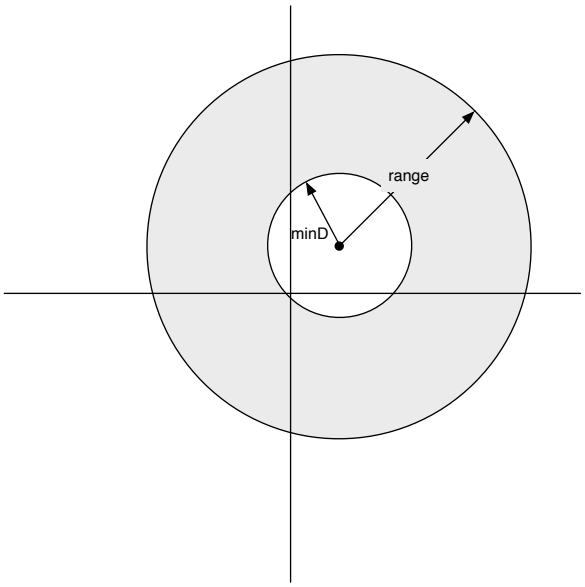
Now this works fine, if you assume that the we are always located at the origin. Suppose the ship may be at a location, `ownposition`, other than the origin. We can update our visualization to look something like this:



We now add an argument representing the location of the ship to our `inRange` function:

```
func inRange2(target: Position, ownPosition: Position, range: Distance) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
}
```

But now you realize that you also want to avoid targeting ships if they are too close to yourself. We can update our picture to illustrate the new situation, where we want to target only those enemies that are at least `minD` away from our current position:



As a result, we need to modify our code again:

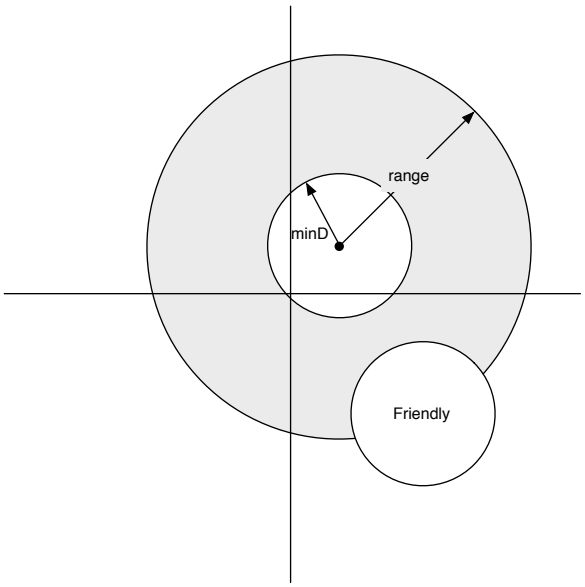
```
let minimumDistance : Distance = 2.0

func inRange3(target: Position, ownPosition: Position, range: Distance) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
          && sqrt(dx * dx + dy * dy) >= minimumDistance
}
```

Finally, you also need to avoid targeting ships that are too close to one of your other ships. We can visualize this by as follows:

Correspondingly, we can add a further argument that represents the location of a friendly ship to our `inRange` function:

```
func inRange4(target: Position, ownPosition: Position,
              friendly: Position, range: Distance) -> Bool {
   let dx = ownPosition.x - target.x
   let dy = ownPosition.y - target.y
   let friendlyDx = friendly.x - target.x
   let friendlyDy = friendly.y - target.y
   return sqrt(dx * dx + dy * dy) <= range
          && sqrt(dx * dx + dy * dy) >= minimumDistance
       && (sqrt(friendlyDx * friendlyDx + friendlyDy * friendlyDy) >= minimumDistance)
}
```

As this code evolves, it becomes harder and harder to maintain. This method expresses a complicated calculation in one big lump of code. Let's try to refactor this into smaller, compositional pieces.



First-class functions

There are different approaches to refactoring this code. One obvious pattern would be to introduce a function that computes the distance between two points; or functions that check when two points are 'close' or 'far away' (for some definition of close and far). In this chapter, however, we'll take a slightly different approach.

The original problem boiled down to defining a function that determined when a point was in range or not. The type of such a function would be something like:

```
func pointInRange(point: Position) -> Bool {
    // Implement method here
}
```

The type of this function is going to be so important, that we're going to give it a separate name:

```
typealias Region = Position -> Bool
```

From now on, the `Region` type will refer to functions from a `Position` to a `Bool`. This isn't strictly necessary, but it can make some of the type signatures that we'll see below a bit easier to digest.

Instead of defining an object or struct to represent regions, we represent a region by a function that determines if a given point is in the region or not. If you're not used to functional programming this may seem strange, but remember: functions in Swift are first-class values! We conciously choose the name `Region` for this type rather than something like `CheckInRegion` or `RegionBlock`. These names suggest that they denote a function type; yet the key philosophy underlying functional programming is that functions are values, no different from structs, Ints or Bools – using a separate naming convention for functions would violate this philosophy.

We will now write several functions that create, manipulate and combine regions.

The first region we define is a `circle`, centered around the origin:

```
func circle(radius: Distance) -> Region {
    return { point in sqrt(point.x * point.x + point.y * point.y) <= radius }
}
```

Note that, given a radius `r`, the call `circle(r)` returns a function. Here we use Swift's notation for closures to construct the function that we wish to return. Given an argument position, `point`, we check that the `point` is in the region delimited by a circle of the given radius centered around the origin.

Of course, not all circles are centered around the origin. We could add more arguments to the `circle` function to account for this. Instead, though, we will write a region transformer:

```
func shift(offset: Position, region: Region) -> Region {
    return { point in
     let shiftedPoint = Position(x: point.x - offset.x, y: point.y - offset.y)
        return region(shiftedPoint)
    }
}
```

The call `shift(offset, region)` moves the region to the right and up by `offet.x` and `offset.y` respectively. How is it implemented? Well we need to return a `Region`, that is a function from points to `Bool`. To do this, we start writing another closure, introducing the point we need to check. From this point, we compute a new point with the coordinates `point.x - offset.x` and `point.y - offset.y`. Finally, we check that this new point is in the original region by passing it as an arguments to the `region` function.

Interestingly, there are lots of other ways to transform existing regions. For instance, we may want to define a new region by inverting a region. The resulting region consists of all the points outside the original region:

```
func invert(region: Region) -> Region {
    return { point in !region(point) }
}
```

We can also write functions that combine existing regions into larger, complex regions. For instance, these two functions take the points that are in both argument regions or either argument region, respectively:

```
func intersection(region1 : Region, region2 : Region) -> Region {
    return { point in region1(point) && region2(point) }
}

func union(region1 : Region, region2 : Region) -> Region {
    return { point in region1(point) || region2(point) }
}
```

Of course, we can use these functions to define even richer regions. The `crop` function takes two regions as argument, `region` and `minusRegion`, and constructs a region with all points that are in the first, but not in the second region.

```
func difference(region: Region, minusRegion: Region) -> Region {
    return intersection(region, invert(minusRegion))
}
```

This example shows how Swift lets you compute and pass around functions no differently than integers or booleans.

Now let's turn our attention back to our original example. With this small library in place, we can now refactor the complicated `inRange` function as follows:

```
func inRange(ownPosition: Position, target: Position,
             friendly: Position, range: Distance) -> Bool {
  let targetRegion = shift(ownPosition, difference(circle(range),
                                        circle(minimumDistance)))
  let friendlyRegion = shift(friendly, circle(minimumDistance))
  let resultRegion = difference(targetRegion, friendlyRegion)
  return resultRegion(target)
}
```

This code defines two regions, `targetRegion` and `friendlyRegion`. The region that we're interested in is computed by taking difference between these regions. By applying this region to the `target` argument, we can compute the desired boolean.

The way we've defined the `Region` type does have its disadvantages. In particular, we cannot inspect how a region was constructed: is it composed of smaller regions? Or is it simply a circle around the origin? The only thing we can do is to check whether or not a given point is within a region or not. If we would want to visualize a region, we would have to sample enough points to generate a (black and white) bitmap.

In later chapters we will sketch an alternative design that will let you answer these questions.

Type-driven development

In the introduction, we mentioned how functional programs take the application of functions to arguments as the canonical way to assemble bigger programs. In this chapter, we have seen a concrete example of this functional design methodology. We have defined a series of functions for describing regions. Each of these functions is not very powerful by itself. Yet together, they can describe complex regions that you wouldn't want to write from scratch.

The solution is simple and elegant. It is quite different from what you might write, had you just refactored the `inRange4` function into separate methods. The crucial design decision we made was how to define regions. Once we chose the `Region` type, all the other definitions followed naturally. The moral of the example is choose your types carefully. More than anything else, types guide the development process.

Chapter 2

Wrapping Core Image

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes.

What this chapter is about

The previous chapter introduced the concept of higher-order function and showed how functions can be passed as arguments to other functions. The example, however, may seem far removed from the 'real' code that you write on a daily basis. In this chapter, we will show how to use higher-order functions to write a small, functional wrapper around an existing, object-oriented API.

Core Image is a powerful image processing framework, but its API can be a bit clunky to use at times. The Core Image API is loosely typed – image filters are configured using key-value-coding. It is all too easy to make mistakes in the type or name of arguments, which can result in run-time errors. The new API we develop will be safe and modular, exploiting types to guarantee the absence of runtime errors.

Don't worry if you're unfamiliar with Core Image or cannot understand all details of the code fragments in this chapter. The goal isn't too build a complete wrapper around Core Image, but instead to illustrate how concepts from functional programming, such as higher-order functions, can be applied in production code.

The Filter Type

One of the key classes in Core Image is the `CIFilter` class, used to create image filters. When you instantiate a `CIFilter` object, you (almost) always provide an input image via the `kCIInputImageKey` key and then retrieve the filtered result via the `kCIOutputImageKey` key. Then you can use this result as input for the next filter.

In the API we will develop in this chapter, we'll try to encapsulate the exact details of these keys-value pairs and present a safe, strongly-typed API to our users. We define our own `Filter` type as a function that takes an image as parameter and returns a new image:

```
typealias Filter = CIImage -> CIImage
```

This is the base type that we are going to build upon.

Building Filters

Now that we have the `Filter` type defined, we can start defining functions that build specific filters. These are convenience functions that take the parameters needed for a specific filter and construct a value of type `Filter`. These functions will all have the following general shape:

```
func myFilter(/* parameters */) -> Filter
```

Note that the return value, `Filter`, is a function as well. This will help us later on to compose multiple filters to achieve the image effects we want.

To make our lives a bit easier, we'll extend the `CIFilter` class with a convenience initializer and a computed property to retrieve the output image:

```
typealias Parameters = Dictionary<String, AnyObject>

extension CIFilter {

    convenience init(name: String, parameters: Parameters) {
        self.init(name: name)
        setDefaults()
        for (key, value : AnyObject) in parameters {
            setValue(value, forKey: key)
        }
    }

   var outputImage: CIImage { return self.valueForKey(kCIOutputImageKey) as CIImage }

}
```

The convenience initializer takes the name of the filter and a dictionary as parameters. The key-value pairs in the dictionary will be set as parameters on the new filter object. Our convenience initializer follows the Swift pattern of calling the designated initializer first.

The computed property `outputImage` provides an easy way to retrieve the output image from the filter object. It looks up the `kCIOutputImageKey` key in and casts the result to a value of type `CIImage`. By providing this computed property of type `CIImage`, users of our API no longer need to cast the result of such a lookup operation themselves.

Blur

With these pieces in place, we can define our first simple filters. The gaussian blur filter only has the blur radius as parameter. Therefore we can write a blur `Filter` very easily:

```
func blur(radius: Double) -> Filter {
    return { image in
     let parameters : Parameters = [kCIInputRadiusKey: radius, kCIInputImageKey: image]
       let filter = CIFilter(name:"CIGaussianBlur", parameters:parameters)
```

```
        return filter.outputImage
    }
}
```

That's all there is to it. The `blur` function returns a function that takes an argument `image` of type `CIImage` and returns a new image (`return filter.outputImage`). Therefore the return value of the `blur` function conforms to the `Filter` type we have defined previously as `CIImage -> CIImage`.

This example is just a thin wrapper around a filter that already exists in Core Image. We can use the same pattern over and over again to create our own filter functions.

Color Overlay

Let's define a filter that overlays an image with a solid color of our choice. Core Image doesn't have such a filter by default, but we can of course compose it from existing filters.

The two building blocks we're going to use for this is the color generator filter (`CIConstantColorGenerator`) and the source over compositing filter (`CISourceOverCompositing`). Let's first define a filter to generate a constant color plane:

```
func colorGenerator(color: NSColor) -> Filter {
    return { _ in
        let filter = CIFilter(name:"CIConstantColorGenerator",
                              parameters: [kCIInputColorKey: color])
        return filter.outputImage
    }
}
```

This looks very similar to the `blur` filter we've defined above with one notable difference: the constant color generator filter does not inspect its input image. Therefore we don't need to name the image parameter in the function being returned; instead we use an unnamed parameter, `_`, to emphasise that the image argument to the filter we are defining is ignored.

Next, we're going to define the composite filter:

```
func compositeSourceOver(overlay: CIImage) -> Filter {
    return { image in
      let parameters : Parameters = [kCIInputBackgroundImageKey: image, kCIInputImageKey: ov
        let filter = CIFilter(name:"CISourceOverCompositing",
                              parameters: parameters)
        return filter.outputImage.imageByCroppingToRect(image.extent())
    }
}
```

Here we crop the output image to the size of the input image. This is not strictly necessary and depends on how we want the filter to behave. This choice works well in the examples we will cover.

Finally we combine these two filters to create our color overlay filter:

```
func colorOverlay(color: NSColor) -> Filter {
    return { image in
        let overlay = colorGenerator(color)(image)
        return compositeSourceOver(overlay)(image)
    }
}
```

Once again, we return a function that takes an image parameter as its argument. The `colorOverlay` starts by calling the `colorGenerator` filter. The `colorGenerator` filter requires a `color` as its argument and returns a filter; hence the code snippet `colorGenerator(color)` has type `Filter`. The `Filter` type, however, is itself a function from `CIImage` to `CIImage`; we can pass an additional argument of type `CIImage` to `colorGenerator(color)` to compute a new overlay `CIImage`. This is exactly what happens in the definition of `overlay` – we create a filter using the `colorGenerator` function and pass the `image` argument to this filter to create a new image. Similarly, the value returned, `compositeSourceOver(overlay)(image)`, consists of a filter being constructed, `compositeSourceOver(overlay)`, and subsequently being applied to the `image` argument.

Composing Filters

Now that we have a blur and a color overlay filter defined, we can put them to use on an actual image in a combined way: first we blur the image and then we put a red overlay on top. Let's load an image to work on:

```
let url = NSURL(string: "https://lh4.googleusercontent.com/-YCRFnjDOiwk/AAAAAAAAAAI/AAAAAA
let image = CIImage(contentsOfURL: url)
```

Now we can apply both filters to these by chaining them together:

```
let blurRadius = 5.0
let overlayColor = NSColor.redColor().colorWithAlphaComponent(0.2)
let blurredImage = blur(blurRadius)(image)
let overlaidImage = colorOverlay(overlayColor)(blurredImage)
```

Once again, we assemble images by creating a filter, such as `blur(blurRadius)`, and applying the resulting filter to an image.

Function Composition

Of course we could simply combine the two filter calls in the above code in a single expression:

```
let result = colorOverlay(overlayColor)(blur(blurRadius)(image))
```

This becomes unreadable very quickly with all these parentheses involved. A nicer way to do this is to compose filters by defining a custom operator for filter composition. To do so, we'll start by defining a function that composes filters:

```
func composeFilters(filter1: Filter, filter2: Filter) -> Filter {
    return {img in filter1(filter2(img)) }
}
```

The `composeFilters` function takes two argument filters and defines a new filter. This composite filter expects an argument `img` of type `CIImage`, and passes it through both `filter2` and `filter1` respectively. We can use function composition to define our own composite filter like this:

```
let myFilter1 = composeFilters(blur(blurRadius), colorOverlay(overlayColor))
let result1 = myFilter1(image)
```

We can even go one step further to make this even more readable by introducing an operator for filter composition. Granted, defining your own operators all over the place doesn't necessarily contribute to the readability of your code, but filter composition is such a recurring pattern that it makes a lot of sense to do this at this point:

```
infix operator |> { associativity left }

func |> (filter1: Filter, filter2: Filter) -> Filter {
    return {img in filter1(filter2(img))}
}
```

Now we can use the `|>` operator in the same way we have used the `composeFilters` before:

```
let myFilter2 = blur(blurRadius) |> colorOverlay(overlayColor)
let result2 = myFilter2(image)
```

The `|>` notation is borrowed from F#. The filter composition operation that we have defined is an example of function composition. In mathematics, the composition of two functions `f` and `g`, sometimes written `f ∘ g`, defines a new function mapping an input to `x` to `f(g(x))`. This is precisely what our `|>` operator does – it passes an argument image through its two constituent filters.

Theoretical background: currying

In this chapter, we've seen that there are two ways to define a function that takes two arguments. The first style is familiar to most programmers:

```
func add1(x:Int, y : Int) -> Int {
  return x + y
}
```

The `add1` function takes two integer arguments and returns their sum. In Swift, however, we can also define another version of the same function:

```
func add2(x : Int) -> (Int -> Int) {
  return {y in return x + y}
}
```

Here the function `add2` takes one argument, `x`, and returns a closure, expecting a second argument `y`. These two add functions must be invoked differently:

```
add1(1,2)
add2(1)(2)

> 3
```

In the first case, we pass both arguments to `add1` at the same time; in the second, we first pass the first argument `1`, which returns a function, which we apply to the second argument `2`. Both versions are equivalent: we can define `add1` in terms of `add2` and visa versa.

In this fashion, we can always transform a function that expects multiple arguments into a series of functions that each expect one argument. This process is referred to as currying, named after the logician Haskell Curry; we say that `add2` is the curried version of `add1`. The popular functional programming language Haskell, which you may have heard of, is also named after Haskell Curry.

In Swift, we can even leave out some of the parentheses in the type signature of `add2`, and write:

```
func add2(x : Int) -> Int -> Int {
  {y in return x + y}
}
```

The function arrow, `->`, associates to the right. That is to say, you can read the type `A -> B -> C` as `A -> (B -> C)`. Throughout this book, however, we will typically introduce a typealias for functional types (as we did for the `Region` and `Filter` types) or write explicit parentheses.

Discussion

This example illustrates, once again, how we break complex code into small pieces, which can all be reassembled using function application. The goal of this chapter was not to define a complete API around Core Image, but instead to sketch how higher-order functions and function composition can be used in a more practical case study.

Why go through all this effort? The Core Image API is already mature and provides all the functionality you might need. We believe there are several advantages to the API designed in this chapter:

- Safety – using the API we have sketched, it is almost impossible to create runtime errors arising from undefined keys or failed casts.
- Modularity – it is easy to compose filters using the `|>` operator. Doing so allows you to tease apart complex filters into smaller, simpler, reusable components.
- Clarity – even if you have never used Core Image, you should be able to assemble simple filters using the functions we have defined. You don't need to know about special dictionary keys to access the results, such as `kCIOutputImageKey`, or worry about initialising certain keys, such as `kCIInputImageKey` or `kCIInputRadiusKey`. From the types alone, you can almost figure out how to use the API, even without further documentation.

Our API presents a series of functions that can be used to define and compose filters. Any filters that you define are safe to use and reuse. Each filter can be tested and understood in isolation. We believe these are compelling reasons to favor the design sketched here over the original Core Image API.

Chapter 3

Map, filter, reduce

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes.

What this chapter is about

First-class functions are prevalent in Swift's standard library. Functions that take functions as arguments are sometimes called higher-order functions. In this chapter, we will tour some of the higher-order functions functions on Arrays from the Swift standard library. By doing so, we will introduce Swift's generics and show how to write assemble complex computations on arrays.

Introducing generics

Suppose we need to write a function that, given an array of integers, computes a new array, where every integer in the original array has been incremented by one. Such a function is easy to write using a single `for` loop:

```
func incrementArray (xs : [Int]) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(x + 1)
    }
    return result
}
```

Now suppose we also need a function that computes a new array, where every element in the argument array has been doubled. This is also easy to do using a `for` loop:

```
func doubleArray1 (xs : [Int]) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(x * 2)
```

```
    }
    return result
}
```

Both these functions share a lot of code. Can we abstract over the differences and write a single, more general function that captures this pattern? Such a function would look something like this:

```
func computeIntArray (xs : [Int]) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(... \\ something using x)
    }
    return result
}
```

To complete this definition, we need to add a new argument describing how to compute a new integer from `xs[i]` – that is, we need to pass a function as an argument!

```
func computeIntArray (xs : [Int], f : Int -> Int) -> [Int] {
    var result : [Int] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

Now we can pass different arguments, depending on how we want to compute a new array from the old array. The `doubleArray` and `incrementArray` functions become one-liners that call `computeIntArray`:

```
func doubleArray2 (xs : [Int]) -> [Int] {
  return computeIntArray(xs){x in x * 2}
}
```

Note that we are using Swift's syntax for trailing closures again here.

This code is still not as flexible as it could be. Suppose we want to compute a new array of booleans, describing whether the numbers in the original array were even or not. We might try to write something like:

```
func isEvenArray (xs : [Int]) -> [Bool] {
    computeIntArray(xs){x in x % 2 == 0}
}
```

Unfortunately, this code gives a type error. The problem is that our `computeIntArray` function takes an argument of type `Int -> Int`, function that computes a new integer. In the definition of `isEvenArray` we are passing an argument of type `Int -> Bool` which causes the type error.

How should we solve this? One thing we could do is define a new version of `computeIntArray` that expects takes a function argument of type `Int -> Bool`. This might look something like this:

```
func computeBoolArray (xs : [Int], f : Int -> Bool) -> [Bool] {
    let result : [Bool] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

This doesn't scale very well though. What if we need to compute a `String` next? Do we need to define yet another higher-order function, expecting an argument of type `Int -> String`?

Luckily there is a solution to this problem: use generics. The definitions of `computeBoolArray` and `computeIntArray` are identical; the only difference is in the type signature. If we were to define another version, `computeStringArray`, the body of the fuction would be the same again. In fact, the same code will work for any type. What we'd really want to do is write a single generic function once and for all, that will work for every possible type.

```
func genericComputeArray<T> (xs : [Int], f : Int -> T) -> [T] {
    var result : [T] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

The most interesting about this piece of code is its type signature. You may want to read `genericComputeArray<T>` as a family of functions; any choice of the type variable `T` determines a function that takes an array of integers and a function of type `Int -> T` as arguments, and returns an array of type `[T]`.

We can generalize this function even further. There is no reason for this function to operate exclusively on input arrays of type `[Int]`. Abstracting over this yields the type signature of

```
func map<T,U> (xs : [T], f : T -> U) -> [U] {
    var result : [U] = []
    for x in xs
    {
        result.append(f(x))
    }
    return result
}
```

Here we have written a function map that is generic in two dimensions: for any array of Ts and function f : T -> U, it will produce a new array of Us. This map function is even more generic than the genericComputeArray function we have seen so far. In fact, we can define genericComputeArray in terms of map:

```
func computeIntArray<T> (xs : [Int], f : Int -> T) -> [T] {
       return map(xs,f)
}
```

Once again, the definition of the function is not that interesting: given two arguments, xs and f, apply map to (xs,f) and return the result. The types are the most interesting thing about this definition. The genericComputeArray is an instance of the map function, it only has a more specific type.

There is already a map method defined in the Swift standard library in the Array class. Instead of writing map(xs,f) for some array xs and function f, we can call the map function from the Array class by writing xs.map(f). Here is an example definition of the doubleArray function using Swift's built-in map function:

```
func doubleArray3 (xs : [Int]) -> [Int] {
  return xs.map{x in 2 * x}
}
```

The point of this chapter is not to argue that you should define map yourself; we do want to argue that there is no magic involved in the definition of map – you could have defined it yourself!

Filter

The map function is not the only function in Swift's standard Array library that uses generics. In this section we will introduce a few others.

Suppose we have an array containing Strings, representing the contents of a directory:

```
let exampleFiles = ["README.md", "HelloWorld.swift",
                    "HelloSwift.swift", "FlappyBird.swift"]
```

Now suppose we want an array of all the .swift files. This is easy to compute with a simple loop:

```
func getSwiftFiles(files: [String]) -> [String] {
    var result : [String] = []
    for file in files {
        if file.hasSuffix(".swift") {
            result.append(file)
        }
    }
    return result
}
```

We can now use this function to ask for the Swift files in our exampleFiles array:

```
getSwiftFiles(exampleFiles)

> [HelloWorld.swift, HelloSwift.swift, FlappyBird.swift]
```

Of course, we can generalize the `getSwiftFiles` function. For instance, we could pass an additional `String` argument to check against instead of hardcoding the `.swift` extension. We could then use the same function to check for `.swift` or `.md` files. But what if we want to find all the files without a file extension? Or the files starting with the string `"Hello"`?

To perform such queries, we define a general purpose `filter` function. Just as we saw previously with `map`, the `filter` function takes a function as an argument. This function has type `T -> Bool` – for every element of the array this function will determine whether or not it should be included in the result.

```
func filter<T> (xs : [T], check : T -> Bool) -> [T] {
    var result : [T] = []
    for x in xs {
        if check(x) {
            result.append(x)
        }
    }
    return result
}
```

It is easy to define `getSwiftFiles` in terms of `filter`. Just like `map`, the `filter` function is defined in the Array class in Swift's standard library.

Now you might wonder: is there an even more general purpose function that can be used to define both `map` and `filter`? In the last part of this chapter, we will answer that question.

Reduce

Once again, let's consider a few simple functions, before defining a generic function that captures the general pattern.

It is straightforward to define a function that sums all the integers in an array:

```
func sum(xs : [Int]) -> Int {
    var result : Int = 0
    for x in xs {
        result += x
    }
    return result
}

let xs = [1,2,3,4]
sum(xs)

> 10
```

A similar for-loop computes the product of all the integers in an array:

```
func product(xs : [Int]) -> Int {
    var result : Int = 1
    for x in xs {
        result = x * result
    }
    return result
}
```

Similarly, we may want to concatenate all the strings in an array:

```
func concatenate(xs : [String]) -> String {
    var result = ""
    for x in xs {
        result += x
    }
    return result
}
```

Or concatenate all the strings in an array, inserting a separate header line and newline characters after every element:

```
func prettyPrintArray (xs : [String]) -> String {
    var result = "Entries in the array xs:\n"
    for x in xs {
        result = "  " + result + x + "\n"
    }
    return result
}
```

What do all these functions have in common? They all initialize a variable, `result`, with some value. They proceed by iterating over all the elements of the input array `xs`, updating the result somehow. To define a generic function that can capture this pattern there are two pieces of information that we need to abstract over: the initial value assigned to the `result` variable; the function used to update the `result` in every iteration.

With this in mind, we arrive at the following definition for the `reduce` function that captures this pattern:

```
func reduce<A,R>(arr : [A], initialValue: R, combine: (R,A) -> R) -> R {
    var result = initialValue
    for i in arr {
        result = combine(result,i)
    }
    return result
}
```

The type of reduce is a bit hard to read at first. It is generic in two ways: for any input array of type `[A]` it will compute a result of type R. To do this, it needs an initial value of type R (to assign to the `result` variable) and a function `combine : (R,A) -> R` that is used to update the result variable in the body of the for loop. In some functional languages, such as OCaml and Haskell, `reduce` functions are called `fold` or `fold_right`.

We can define every function we have seen in this chapter so far using `reduce`. Here are a few examples:

```
func sumUsingReduce (xs : [Int]) -> Int {
  return reduce(xs, 0) {result, x in result + x}
  }

func productUsingReduce (xs : [Int]) -> Int {
  return reduce(xs, 1) {result, x in result * x}
  }

func concatUsingReduce (xs : [String]) -> String {
  return reduce (xs, "") {result, x in result + x}
  }
```

In fact, we can even redefine map and `filter` using reduce:

```
func mapUsingReduce<T,U> (xs : [T], f : T -> U) -> [U] {
  return reduce (xs,[]) {result, x in result + [f(x)]}
}

func filterUsingReduce<T> (xs : [T], check : T -> Bool) -> [T]{
  return reduce(xs, []) {result, x in check(x) ? result + [x] : result}
}
```

This shows how the `reduce` function captures a very common programming pattern: iterating over an array to compute a result.

Putting it all together

To conclude this section, we will give a small example of map, `filter` and `reduce` in action.

Suppose we have the following `struct` definition, consisting of a city's name and population (measured in thousands-of-inhabitants):

```
struct City {
    let name : String
    let population : Int
}
```

We can define several example cities:

```
let paris = City(name: "Paris", population: 2243)
let madrid = City(name: "Madrid", population: 3216)
let amsterdam = City(name: "Amsterdam", population: 811)
let berlin = City(name: "Berlin", population: 3397)

let cities = [paris, madrid, amsterdam, berlin]
```

Now suppose we would like to print a list of cities with at least one million inhabitants, together with their total population. We can define a helper function that scales up the inhabitants:

```
func scale(city : City) -> City {
    return City(name: city.name, population: city.population * 1000)
}
```

Now we can use all the ingredients we have seen in this chapter to write the following statement:

```
cities.filter({city in city.population > 1000})
      .map(scale)
      .reduce("City : Population",
         {result, c in result + "\n" + "\(c.name) : \(c.population)" }
       )
```

We start by filtering out those cities that have less than one million inhabitants; we then map our `scale` function over the remaining cities; and finally, we compute a `String` with a list of city names and populations using the `reduce` function. Here we use the `map`, `filter` and `reduce` definitions from the Array class in Swift's standard library. As a result, we can chain together the results of our maps and filters nicely. The `cities.filter(..)` expression computes an Array, on which we call `map`; we call `reduce` on the result of this call to obtain our final result.

Chapter 4

QuickCheck

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes.

In recent years, testing has become much more prevalent in Objective-C. Many popular libraries are now tested automatically with continuous integration tools. The standard framework for writing unit tests is XCTest. In addition, a lot of third-party frameworks are available (such as Specta, Kiwi and FBSnapshotTestCase), and a number of frameworks are currently being developed in Swift.

All of these frameworks follow a similar pattern: they typically consist of some fragment of code, together with an expected result. The code is then executed; its result is then compared to the expected result mentioned in the test. Different libraries test at different levels: some might test individual methods, some test classes and some perform integration testing (running the entire app). In this chapter, we will build a small library for property-based testing of Swift code.

When writing unit tests, the input data is static and defined by the programmer. For example, when unit-testing an addition method, we might write a test that verifies that `1 + 1` is equal to 2. If the implementation of addition changes in such a way that this property is broken, the test will fail. More generally, however, we might test that addition is commutative, or in other words, that `a + b` is equal to `b + a`. To test this, we might write a test case that verifies that `42 + 7` is equal to `7 + 42`.

In this chapter, we'll build Swift port (a part of) QuickCheck,[1] a Haskell library for random testing. Instead of writing individual unit tests that each test a function is correct for some particular input, QuickCheck allows you to describe abstract properties of your functions and generate tests to verify them.

This is best illustrated with an example. Suppose we want to verify that plus is commutative. To do so, we start by writing a function that checks where `x + y` is equal to `y + x` for two integers `x` and `y`:

```
func plusIsCommutative(x : Int, y : Int) -> Bool {
    return x + y == y + x
}
```

Checking this statement with QuickCheck is as simple as calling the `check` function:

[1] http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.47.1361 "QuickCheck: A Lightweight Tool for Random Testing of Haskell Programs"

```
check("Plus should be commutative", plusIsCommutative)

> "Plus should be commutative" passed 100 tests.
> ()
```

The `check` function works by calling the `plusIsCommutative` function with two random integers, over and over again. If the statement isn't true, it will print out the input that caused the test to fail. The key insight here is that we can describe abstract properties of our code (like commutativity) using functions that return a `Bool` (like `plusIsCommutative`). The `check` function now uses this property to generate unit tests; giving much better code coverage than you could achieve using hand-written unit tests.

Of course, not all tests pass. For example, we can define a statement that describes that the subtraction is commutative:

```
func minusIsCommutative(x : Int, y : Int) -> Bool {
    return x - y == y - x
}
```

Now, if we run QuickCheck on this function, we will get a failing test case:

```
check("Minus should be commutative", minusIsCommutative)

> "Minus should be commutative" doesn't hold: (5, 4)
> ()
```

Using Swift's syntax for trailing closures, we can also write tests directly, without defining the property (such as `plusIsCommutative` or `minusIsCommutative`) separately:

```
check("Additive identity") {(x : Int) in x + 0 == x }

> "Additive identity" passed 100 tests.
> ()
```

Building QuickCheck

In order to build this library, we will need to do a couple of things. First, we need a way to generate random values for different kinds of types. Then, we need to implement the `check` function, which will feed our test with random values a number of times. Should a test fail, then we'd like to make the input smaller. For example, if our test fails on an array with 100 elements, we'll try to make it smaller and see if the test still fails. Finally, we'll need to do some extra work to make sure our check function works on types that have generics.

Generating Random Values

First, let's define a protocol that knows how to generate arbitrary values. As the return type, we use `Self`, which is the type of the class that implements it.

```
protocol Arbitrary {
    class func arbitrary() -> Self
}
```

First, let's write an instance for `Int`. We use the `arc4random` function from the standard library and convert it into an `Int`. Note that this only generates positive integers. A real implementation of the library would generate negative integers as well, but we'll try to keep things simple in this chapter.

```
extension Int : Arbitrary {
    static func arbitrary() -> Int {
        return Int(arc4random())
    }
}
```

Now we can generate random integers like this:

```
Int.arbitrary()
```

```
> 3737486643
```

To generate random strings, we need to do a little bit more work. First, we generate a random length x between 0 and 100. Then, we generate x random characters, and reduce them into a string. Note that we currently only generate capital letters as random characters, a real implementation would generate any kind of character.

```
extension Character : Arbitrary {
  static func arbitrary() -> Character {
    return Character(UnicodeScalar(random(from: 65, to:90)))
  }

  func smaller() -> Character? { return nil }
}

extension String : Arbitrary {
    static func arbitrary() -> String {
        let randomLength = random(from: 0, to: 100)
     let randomCharacters = repeat(randomLength) { _ in Character.arbitrary() }
        return reduce(randomCharacters, "", +)
    }

}
```

We can call it in the same way as we generate random `Int`s, except, that we call it on the `String` class:

```
String.arbitrary()
```

```
> MECVLYWJLYSABUXGDSUUGBYEIMVOBGIOQMLHDX
```

Implementing the check function

Now we are ready to implement a first version of our check function. The check1 function consists of a simple loop that generates random input for the argument property in every iteration. If a counterexample is found, it is printed and the function returns; if no counterexample is found, the check1 function reports the number of successful tests that have passed. (Note that we called the function check1, because we'll write the final version a bit later).

```
func check1<A: Arbitrary>(message: String, prop: A -> Bool) -> () {
    for _ in 0..<numberOfIterations {
        let value = A.arbitrary()
        if !prop(value) {
            println("\"\(message)\" doesn't hold: \(value)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

Here's how we can use this function to test properties:

```
check1("Additive identity") {(x : Int) in x + 0 == x }
```

```
> "Additive identity" passed 100 tests.
> ()
```

Making values smaller

If we run our check1 function on strings, we might get quite a long failure message:

```
check1("Every string starts with Hello") {(s: String) in s.hasPrefix("Hello")}
```

```
> "Every string starts with Hello" doesn't hold: ILKJFJGCDXTDMKKRIAEBJKVCDXOGKQRSUGSADJESKV
> ()
```

Ideally, we'd like our failing input to be a short as possible. In general, the smaller the counterexample, the easier it is to spot which piece of code is causing the failure. In this example, the counterexample is still pretty easy to understand – but this may not always be the case! Imagine a complicated property on arrays or dictionaries that fails for some unclear reason – debugging is much easier with a minimal counterexample. In principle, the user could try to trim the input that triggered the failure and try rerunning the test; rather than place the burden on the user, however, we will automate this process.

To do so, we will make an extra protocol called Smaller, which does only one thing: it tries to shrink the counterexample.

```
protocol Smaller {
    func smaller() -> Self?
}
```

Note that the return type of the `smaller` function is marked as optional. There are cases when it is not clear how to shrink test data any further. For example, there is no obvious way to shrink an empty array. We will return `nil` in that case.

In our instance for integers we just try to divide the integer by two until we reach zero:

```
extension Int : Smaller {
    func smaller() -> Int? {
        return self == 0 ? nil : self / 2
    }
}
```

We can now test our instance:

```
100.smaller()
```

```
> Optional(50)
```

For strings, we just drop the first character (unless the string is empty).

```
extension String : Smaller {
    func smaller() -> String? {
       return self.isEmpty ? nil : self[startIndex.successor()..<endIndex]
    }
}
```

To use the `Smaller` protocol in the `check` function, will need the ability to shrink any test data generated by our `check` function. To do so, we will redefine our `Arbitrary` protocol to extend the `Smaller` protocol:

```
protocol Arbitrary : Smaller {
    class func arbitrary() -> Self
}
```

Repeatedly shrinking

We can now redefine our `check` function to shrink any test data that triggers a failure. To do this, we use the `iterateWhile` function that takes a condition, an initial value and repeatedly applies a function as long as the condition holds.

```
func iterateWhile<A>(condition: A -> Bool, initialValue: A, next: A -> A?) -> A {
    if let x = next(initialValue) {
        if condition(x) {
           return iterateWhile(condition,x,next)
        }
    }
    return initialValue
}
```

Using `iterateWhile` we can now repeatedly shrink counterexamples that we uncover during testing:

```
func check2<A: Arbitrary>(message: String, prop: A -> Bool) -> () {
    for _ in 0..<numberOfIterations {
        let value = A.arbitrary()
        if !prop(value) {
         let smallerValue = iterateWhile({ value in !prop(value) }, value) {
              $0.smaller()
            }
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

Arbitrary Arrays

Currently, our `check2` function only supports `Int` and `String` values. While we are free to define new extensions for other types, such as `Bool`, things get more complicated when we want to generate arbitrary arrays. As a motivating example, let's write a functional version of QuickSort:

```
func qsort(var array: [Int]) -> [Int] {
    if array.isEmpty { return [] }
    let pivot = array.removeAtIndex(0)
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    return qsort(lesser) + [pivot] + qsort(greater)
}
```

We can also try to write a property to check our version of QuickSort against the built-in sort function:

```
check2("qsort should behave like sort", { (x: [Int]) in return qsort(x) == x.sorted(<) })
```

However, the compiler warns us that `[Int]` doesn't conform to the `Arbitrary` protocol. In order to implement `Arbitrary`, we first have to implement `Smaller`. As a first step, we provide a simple definitian that drops the first element in the array:

```
extension Array : Smaller {
    func smaller() -> [T]? {
     return self.isEmpty ? nil : Array(self[startIndex.successor()..<endIndex])
    }
}
```

We can also write a function that generates an array of arbitrary length for any type that conforms to the `Arbitrary` protocol:

```
func arbitraryArray<X: Arbitrary>() -> [X] {
  let randomLength = Int(arc4random() % 50)
  return repeat(randomLength) {_ in return X.arbitrary() }
}
```

Now what we'd like to do is define an extension that uses the `abritraryArray` function to give the desired `Arbitrary` instance for arrays. However, to define an instance for `Array`, we also need to make sure that the element type of the array is also an instance of `Arbitrary`. For example, in order to generate an array of random numbers, we first need to make sure that we can generate random numbers. Ideally, we would write something like this, saying that the elements of an array should also conform to the arbitrary protocol:

```
extension Array<T: Arbitrary> : Arbitrary {
    static func arbitrary() -> [T] {
        ...
    }
}
```

Unfortunately, it is currently not possible to express this restriction as a type constraint, making it impossible to write an extension that makes `Array` conform to the `Arbitrary` protocol. Instead we will modify the `check2` function.

The problem with the `check2<A>` function was that it required the type A to be `Arbitrary`. We will drop this requirement, and instead require the necessary functions, `smaller` and `arbitrary`, to be passed as arguments.

We start by defining an auxiliary struct that contains the two functions we need:

```
struct ArbitraryI<T> {
    let arbitrary : () -> T
    let smaller: T -> T?
}
```

We can now write a helper function that takes such an `ArbitraryI` struct as an argument. The definition of `checkHelper` closely follows the `check2` function we saw previously. The only difference between the two is where the `arbitrary` and `smaller` functions are defined. In `check2` these were constraints on the generic type, `<A : Arbitrary>`; in `checkHelper` they are passed explicitly in the `ArbitraryI` struct:

```
func checkHelper<A>(arbitraryInstance: ArbitraryI<A>,
                    prop: A -> Bool, message: String) -> () {
    for _ in 0..<numberOfIterations {
        let value = arbitraryInstance.arbitrary()
        if !prop(value) {
        let smallerValue = iterateWhile({ !prop($0) }, value, arbitraryInstance.smaller)
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

This is a standard technique: instead of working with functions defined in a protocol, we pass the required information as an argument explicitly. By doing so, we have a bit more flexibility. We no longer rely on Swift to infer the required information, but have complete control over this ourself.

We can redefine our `check2` function to use the `checkHelper` function. If we know that we have the desired `Arbitrary` definitions, we can wrap them in the `ArbitraryI` struct and call `checkHelper`:

```
func check<X : Arbitrary>(message: String, prop : X -> Bool) -> () {
   let instance = ArbitraryI(arbitrary: { X.arbitrary() }, smaller: { $0.smaller() })
    checkHelper(instance, prop, message)
}
```

If we have a type for which we cannot define the desired `Arbitary` instance, like arrays, we can overload the `check` function and construct the desired `ArbitraryI` struct ourself:

```
func check<X : Arbitrary>(message: String, prop : [X] -> Bool) -> () {
    let instance = ArbitraryI(arbitrary: arbitraryArray,
                              smaller: { (x: [X]) in x.smaller() })
    checkHelper(instance, prop, message)
}
```

Now, we can finally run `check` to verify our QuickSort implementation. Lots of random arrays will get generated and passed to our test.

```
check("qsort should behave like sort", { (x: [Int]) in return qsort(x) == x.sorted(<) })
```

```
> "qsort should behave like sort" passed 100 tests.
> ()
```

Next steps

This library is far from complete, but already quite useful. There are a couple of obvious things that could be improved:

- The shrinking is naive. For example, in the case of arrays, we currently remove the first element of the array. However, we might also choose to remove a different element, or make the elements of the array smaller (or do all of that). The current implementation returns an optional shrinked value, whereas we might want to generate a list of values. In a later chapter we will see how to generate a lazy list of results, we could use the same technique here.
- The Arbitrary instances are quite simple. For different datatypes, we might want to have more complicated arbitrary instances. For example, when generating arbitrary enum values, we might want to generate certain cases with different frequencies. We might also want to generate constrained values (for example, what if we want to test a function that expects sorted arrays?). When writing multiple `Arbitrary` instances, we might want to define some helper functions that aid us in writing these instances.

- We might want to classify the generated test data. For example, if we generate a lot of arrays of length 1, we could classify this as a ‘trivial’ test case. The Haskell library has support for classification, these ideas could be ported directly.

There are many other small and large things that could be improved to make this into a full library.

Chapter 5

Optionals

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes.

What this chapter is about

Swift's optional types can be used to represent values that may be missing or computations that may fail. This chapter describes Swift's optional types, how to work with them effectively, and how they fit well with the functional programming paradigm.

Case study: Dictionaries

Besides arrays, Swift has special support for working with dictionaries. A dictionary is collection of key-value pairs, providing an efficient way to find the value associated with a certain key. The syntax for creating dictionaries is similar to arrays:

```
let cities = ["Paris" : 2243, "Madrid" : 3216, "Amsterdam" : 881, "Berlin" : 3397]
```

Similar to the previous example in Chapter 3, this dictionary stores the population of several European cities. In this example, the key `"Paris"` is associated with the value `2243`; that is, Paris has about `2243000` inhabitants.

Like arrays, the `Dictionary` type is generic. The type of dictionaries is generic in two arguments: the types of the stored keys and values. In our example, the city dictionary has type `Dictionary<String,Int>`. There is also a shorthand notation, `[String : Int]`.

We can lookup the value associated with a key using the same notation as array indexing:

```
let madridPopulation : Int = cities["Madrid"]
```

This example, however, does not type check. The problem is that the key `"Madrid"` may not be in the `cities` dictionary – and what value should be returned if it is not? We cannot guarantee that the dictionary lookup operation always returns an `Int` for every key. Swift's optional types track the possibility of failure. The correct way to write the example above would be:

```
let madridPopulation : Int? = cities["Madrid"]
```

Instead of having type `Int`, the `madridPopulation` example has type the optional type `Int?`. A value of type `Int?` is either an `Int` or a special 'null' value, `nil`.

We can check whether or not the lookup was successful as follows:

```
if madridPopulation != nil {
  println("The population of Madrid is \(madridPopulation! * 1000)")
}
else {
  println("Unknown city: Madrid")
}
```

If `madridPopulation` is not `nil`, the then branch is executed. To refer to the underlying `Int`, we write `madridPopulation!`. The post-fix `!` operator forces an optional to a non-optional type. To compute the total population of Madrid, we force the optional `madridPopulation` to an `Int` and mutiply by `1000`.

Swift has a special optional binding mechanism, that lets you avoid writing the `!` suffix. We can combine the definition of `madridPopulation` and the check above into a single statement:

```
if let madridPopulation = cities["Madrid"] {
  println("The population of Madrid is \(madridPopulation * 1000)")
}
else {
  println("Unknown city: Madrid")
}
```

If the lookup, `cities["Madrid"]`, is succesful we can use the variable `madridPopulation : Int` may be used in the then branch. Note that we no longer need to explicitly use the forced unwrapping operator.

Given the choice, we'd recommend using option binding over forced unwrapping. Forced unwrapping may crash if you have a `nil` value; option binding encourages you to handle exceptional cases explicitly, avoiding run-time errors. Unchecked usage of the forced unwrapping of optional types is a bad code smell. Similarly, Swift's mechanism for implicitly unwrapped optionals is also unsafe and should be avoided whenever possible.

Combining optional values

Swift's optional values make the possibility of failure explicit. This can be cumbersome, especially when combining several optional results. There are several techniques to facilitate the use of optionals.

Optional chaining

First of all, Swift has a special mechanism, optional chaining, for selecting methods or attributes in nested classes or structs. Consider the following (fragment of) a model for processing customer orders:

```
struct Order {
    let orderNumber : Int
    let person : Person?
    ...
}

struct Person {
    let name : String
    let address : Address?
    ...
}

struct Address{
    let streetName : String
    let city : String
    let state : String?
    ...
}
```

Given an `Order`, how can we find the state of the customer? We could use the explicit unwrapping operator:

```
order.person!.address!.state!
```

Doing so, however, may cause run-time exceptions if any of the intermediate data is missing. It would be much safer to use option binding:

```
if let myPerson = order.person in {
  if let myAddress = myPerson.address in {
    if let myState = myAddress.state in {
    ...
```

But this is rather verbose. Using optional chaining, this example would become:

```
if let myState = order.person?.address?.state? {
  print("This order will be shipped to \(myState\)")
}
else {
  print("Unknown person, address, or state.")
```

Instead of forcing the unwrapping of intermediate types, we use the question mark operator to try and unwrap the optional types. When any of the component selections fails, the whole chain of selection statements returns `nil`.

Maps and monads

The ? operator lets us select methods or fields of optional values. There are plenty of other examples, however, where you may want to manipulate an optional value, if it exists, and return `nil` otherwise. Consider the following example:

```
func incrementOptional (maybeX : Int?) -> Int? {
  if let x = maybeX {
    return x + 1
  }
  else {
    return nil
  }
}
```

The `incrementOptional` example behaves similarly to the ? operator: if the optional value is nil, the result is nil; otherwise there some computation is performed.

We can generalize both `incrementOptional` and the ? operator and define a `map` function. Rather than only increment a value of type `Int?`, as we did in `incrementOptional`, we pass the operation we wish to perform as an argument to the `map` function:

```
func map<T,U> (maybeX : T?, f : T -> U) -> U? {
  if let x = maybeX {
    return f(x)
  }
  else {
    return nil
  }
}
```

This `map` function takes two arguments: an optional value of type `T?` and a function `f` of type `T -> U`. If the optional value is non-nil, it applies `f` to it and returns the result; otherwise the `map` function returns nil. This `map` function is part of the Swift standard library.

Using `map` we write the `incrementOptional` function as:

```
func incrementOptional2 (maybeX : Int?) -> Int? {
    return maybeX.map{x in x + 1}
}
```

Why is this function called `map`? What does it have to do with array computations? To see this, it helps to expand some of the shorthand notation that Swift uses. Optional types, such as `Int?`, can also be written out explicitly as `Optional<Int>`, in the same way that we can write `Array<T>` rather than `[T]`. If we now state the types the `map` function on arrays and optionals, the similarity becomes apparent:

```
func mapOptional<T,U> (maybeX : Optional<T>, f : T -> U) -> Optional<U>

func mapArray<T,U> (xs : Array<T>, f : T -> U) -> Array<U>
```

Both `Optional` and `Array` are type constructors that expect a generic type argument. For instance, `Array<T>` or `Optional<Int>` are valid types, but `Array` by itself is not. Both these `map` functions take two arguments: the structure being mapped and a function `f` of type `T -> U`. The `map` functions use `f` to transform all the values of type `T` to values of type `U` in the argument array or optional. Type constructors, such as optionals or arrays, that support a `map` operation are sometimes referred to as functors.

Of course, we can also use `map` to project fields or methods from optional structs and classes, similar to the `?` operator.

The `map` function shows one way to manipulate optional values, but many others exist. Consider the following example:

```
let x : Int? = 3
let y : Int? = nil
let z : Int? = x + y
```

This program is not accepted by the Swift compiler. Can you spot the error?

The problem is that addition only works on `Int` values, rather than the optional `Int?` values we have here. To resolve this, we would have to introduce nested `if` statements as follows:

```
func addOptionals (maybeX : Int?, maybeY : Int?) -> Int? {
  if let x = maybeX {
    if let y = maybeY {
        return x + y}
    }
  return nil
}
```

This may seem like a contrived example, but manipulating optional values can happen all the time. Suppose we have the following dictionary, associating capital cities with their inhabitants:

```
let capitals = ["France" : "Paris", "Spain" : "Madrid", "The Netherlands" : "Amsterdam", "Bel
```

We might want to compose this dictionary with the `cities` dictionary we saw previously, to write a function that computes the number of inhabitants in a countries capital city, if we have the information necessary to compute this:

```
func populationOfCapital (country : String) -> Int? {
  if let capital = capitals[country] {
    if let population = cities[capital] {
      return population * 1000
    }
  }
  return nil
}
```

The same pattern pops up again, repeatedly checking if an optional exists, and continuing with some computation when it does. In a language with first-class functions, like Swift, we define a custom operator that captures this pattern:

```
infix operator >>= {}

func >>= <U,T> (maybeX : T?, f : T -> U?) -> U? {
    if let x = maybeX
    {
        return f(x)
    }
    else
    {
        return nil
    }
}
```

The >>= operator checks whether some optional value is non-nil. If it is, we pass it on to the argument function `f`; if the optional argument is nil, the result is also nil.

Using this operator, we can now write our examples as follows:

```
func addOptionals2 (maybeX : Int?, maybeY : Int?) -> Int? {
    return maybeX >>= {x in
            maybeY >>= {y in
            x + y}}
  }

func populationOfCapital2 (country : String) -> Int? {
    return  capitals[country] >>= {capital in
            cities[capital] >>= {population in
            return population * 1000}}
}
```

The choice of operator, >>=, name is no coincidence. Swift's optional types are an example of a monad, similar to Haskell's `Maybe` type. A monad is a type constructor `M` that supports a pair of functions with the following types:

```
  func return<U> (x : U) -> M<U>

  @infix func >>= <U,T> (x : M<T>, f : T -> M<U>) -> M<U>
```

Although we haven't defined the `return` function for optionals, it is trivial to do so.

We do not want to advocate that >>= is 'right' way to combine optional values, or that you need to understand monads to work with Swift's optionals. Instead we hope to show that optionals are not a new idea. They have successfully been used in other languages for many years. The optional binding mechanism captures just the right pattern for writing most functions over optionals – and for good reason: it corresponds to the >>= operation of the optional monad.

Why Optionals?

What's the point of introducing an explicit optional type? For programmers used to Objective C, working with optional types may seem strange at first. The Swift type system is rather rigid: whenever

we have an optional type, we have to deal with the possibility of it being `nil`. We have had to write new functions like `map` to manipulate optional values. In Objective C, you have more flexibility. For instance, when we translate the example above to Objective C, there is no compiler error:

```
- (int)populationOfCapital:(NSString *)country {
    return [self.cities[self.capitals[country]] intValue] * 1000;
}
```

We can pass in the nil for the name of a country, and we get back a result of `0.0`. Everything is fine. In many languages without optionals, null pointers are a source of danger. Much less so in Objective-C. In Objective-C, you can safely send messages to nil, and depending on the return type, you either get nil, 0, or similar "zero-like" values. Why change this behavior in Swift?

The choice for an explicit optional type fits with the increased static safety of Swift. A strong type system catches errors before code is executed; automatic memory allocation and garbage collection limits the possibility for memory leaks; an explicit optional type helps protects you from unexpected crashes arising from nil values.

The default "zero-like" behaviour employed by Objective-C has its drawbacks. You may want to distinguish between failure (a key is not in the dictionary) and success-returning-zero (a key is in the dictionary, but associated with 0). Furthermore, the behavior is not available to all types: you cannot have an integer that might be nil without wrapping it in a class like `NSNumber`.

While it is safe in Objective-C to send messages to nil, it is often not safe to use them. Let's say we want to create an attributed string. If we pass in nil as the argument for `country`, the `capital` will also be nil, but `NSAttributedString` will crash when trying to initialize it with a nil value.

```
- (NSAttributedString *)attributedCapital:(NSString*)country {
    NSString *capital = self.capitals[country];
    return [[NSAttributedString alloc] initWithString:capital
                                            attributes:self.capitalAttributes];
}
```

While crashes like that don't happen too often, almost every developer had code like this crash. Most of the time, these crashes are detected during debugging, but it is very possible to ship code without noticing that in some cases a variable might unexpectedly be nil. Therefore, many programmers use asserts to verify this behavior. For example, we can add a `NSParameterAssert` to make sure we crash quickly when the `country` is nil:

```
- (NSAttributedString *)attributedCapital:(NSString*)country {
    NSParameterAssert(country);
    NSString *capital = self.capitals[country];
    return [[NSAttributedString alloc] initWithString:capital
                                            attributes:self.capitalAttributes];
}
```

Now, when we pass in a country value that is nil, the assert fails immediately, and we are almost certain to hit this during debugging. But what if we pass in a `country` value that doesn't have a matching key in `self.capitals`? This is much more likely, especially when `country` comes from user input. In that case, `capital` will be nil and our code will still crash. Of course, this can be fixed

easily enough. The point is, however, that it is easier to write robust code using nil in Swift than in Objective-C.

Finally, using these assertions is inherently non-modular. Suppose we implement a `checkCountry` method, that checks that a non-nil `NSString*` is supported. We can incorporate this check easily enough:

```
- (NSAttributedString *)attributedCapital:(NSString*)country {
    NSParameterAssert(country);
    if (checkCountry(country))
    ...
}
```

Now the question arises: should the `checkCountry` function also assert that its argument is non-nil. On the one hand, it should not: we have just performed the check in the `attributedCapital` method. On the other hand, if the `checkCountry` function only works on non-nil values, we should duplicate the assertion. We are forced to choose between exposing an unsafe interface or duplicating assertions.

In Swift, things are quite a bit better. Function signatures using optionals explicitly state which values may be nil. This is invaluable information when working with other people's code. A signature like the following provides a lot of information:

```
func attributedCapital(country : String) -> NSAttributedString?
```

Not only are we warned about the possibility of failure, but we know that we must pass a `String` as argument – and not a nil value. A crash like the one we described above will not happen. Furthermore, this is information checked by the compiler. Documentation goes out of date easily; you can always trust function signatures.

Chapter 6

Diagrams

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes.

In this chapter, we'll look at a functional way to describe diagrams, and how to draw them with Core Graphics. By wrapping Core Graphics with a functional layer, we get an API that's simpler and more composable.

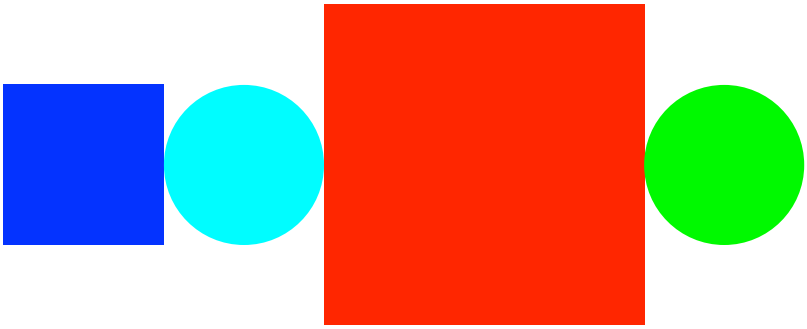
Drawing squares and circles

Imagine drawing the following diagram:

In Core Graphics, we could achieve this drawing with the following command:

```
[[NSColor blueColor] setFill]
CGContextFillRect(context, CGRectMake(0.0,37.5,75.0,75.0))
[[NSColor redColor] setFill]
CGContextFillRect(context, CGRectMake(75.0,0.0,150.0,150.0))
[[NSColor greenColor] setFill]
CGContextFillEllipseInRect(context, CGRectMake(225.0,37.5,75.0,75.0))
```

This is nice and short, but it is a bit hard to maintain. For example, what if we wanted to add an extra circle:



We would need to add the code for drawing a rectangle, but also update the drawing code to move some of the other objects to the right. In Core Graphics, we always describe how to draw things. In this chapter, we'll build a library for diagrams that allows us to express what we want draw. For example, the first diagram can be expressed like this:

```
let blueSquare = square(side: 1).fill(NSColor.blueColor())
let redSquare = square(side: 2).fill(NSColor.redColor())
let greenCircle = circle(radius: 1).fill(NSColor.greenColor())
let example1 = blueSquare ||| redSquare ||| greenCircle
```

And adding the second circle is as simple as changing the last line of code:

```
let example2 = blueSquare ||| circle(radius: 1).fill(NSColor.cyanColor()) ||| redSquare |||
```

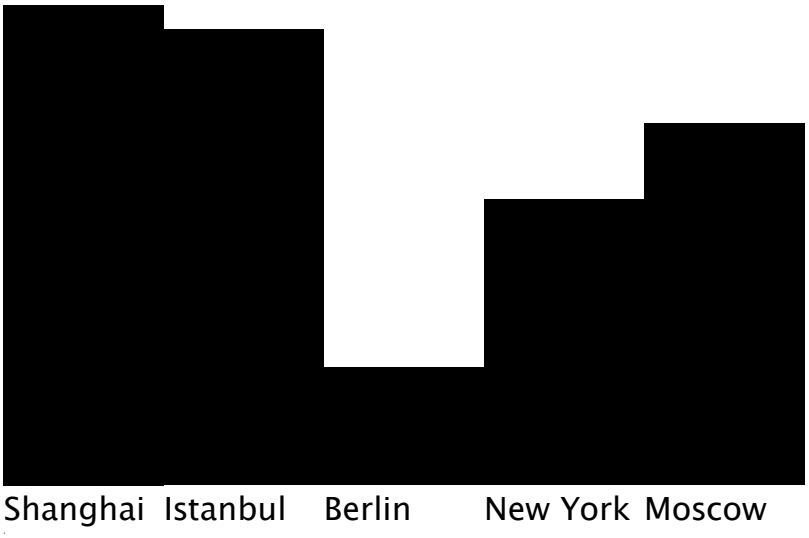
The code above first describes a blue square, with a relative size of 1. The red square is twice as big (it has a relative size of 2), and we compose the diagram by putting the squares and the circle next to each other with the `|||` operator. Changing this diagram is very simple, and there's no need to worry about calculating frames or moving things around. The examples describe what should be drawn, not how it should be drawn.

One of the techniques we'll use in this chapter is building up an intermediate structure of the diagram. Instead of executing the drawing commands immediately, we build up a data structure that describes the diagram. This is a very powerful technique, as it allows us to inspect the data structure, modify it and convert it into different formats.

As a more complex example of a diagram generated by the same library, here's a bar graph:

We can write a `barGraph` function that takes a list of names (the keys) and values (the relative heights of the bars). For each value, we draw a rectangle, and horizontally concatenate them with the `hcat` function. For each name, we draw it as text, also horizontally concatenate them using `hcat`, and finally put the bars and the text below each other using the `---` operator:

```
func barGraph(input: [(String,Double)]) -> Diagram {
    let values : [CGFloat] = input.map { CGFloat($0.1) }
    let bars =  hcat(normalize(values).map { (x: CGFloat) -> Diagram in
     return rect(width: 1, height: 3*x).fill(NSColor.blackColor()).alignBottom()
    })
    let labels = hcat(input.map { x in
     return text(width: 1, height: 0.3, text: x.0).fill(NSColor.cyanColor()).alignTop()
    })
```




```
    return bars --- labels
}
let cities = ["Shanghai": 14.01, "Istanbul": 13.3, "Moscow": 10.56, "New York": 8.33, "Berlin
let example3 = barGraph(cities.keysAndValues)
```

The Core Data Structures

In our library, we'll draw three kinds of things: ellipses, rectangles and text. Using enums, we can define a datatype for that:

```
enum Primitive {
    case Ellipsis
    case Rectangle
    case Text(String)
}
```

It would be very possible to extend this with other primitives, such as images or more complex shapes.

Diagrams are defined using an enum as well. First, a diagram could be a primitive, which has a size and is either an ellipsis, a rectangle or text. Note that we call it `Prim` because, at the time of writing, the compiler gets confused by a case that has the same name as another enum.

```
case Prim(CGSize, Primitive)
```

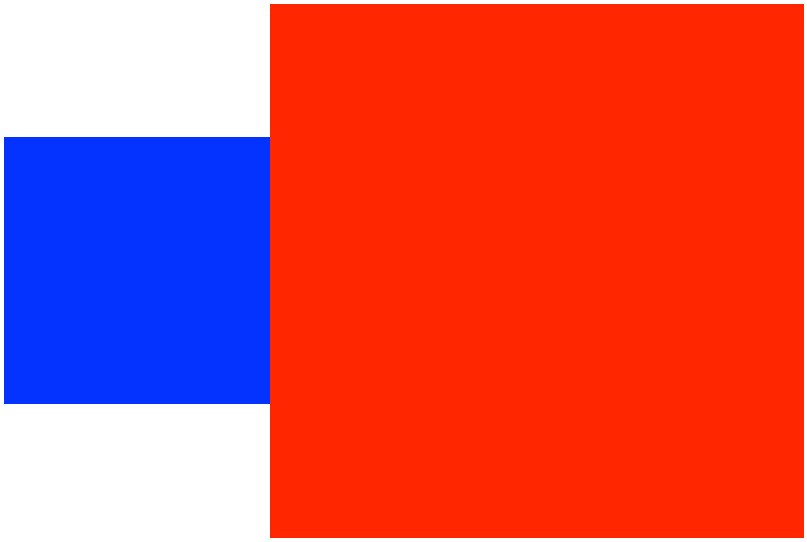
Then, we have cases for diagrams that are beside each other (horizontally) or below each other (vertically). Note how a `Beside` diagram is defined recursively: it consists of two diagrams next to each other.

```
case Beside(Diagram,Diagram)
case Below(Diagram,Diagram)
```

To style diagrams, we'll add a case for attributed diagrams. This allows us to set the fill color (for example, for ellipses and rectangles). We'll define the `Attribute` type later.

```
case Attributed(Attribute,Diagram)
```

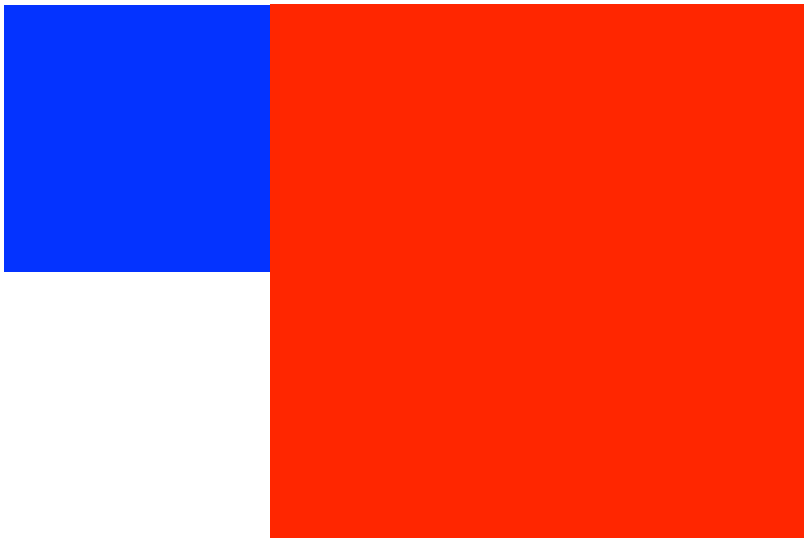
The last case is for alignment. Suppose we have a small and a large rectangle that are next to each other. By default, the small rectangle gets centered vertically:



But by adding a case for alignment we can control the alignment of smaller parts of the diagram.

```
case Align(Vector2D, Diagram)
```

For example, here's a diagram that's top-aligned:



It is drawn using the following code:

```
Diagram.Align(Vector2D(x: 0.5,y: 1), blueSquare) ||| redSquare
```

Unfortunately, in the current version of Swift, recursive datatypes are not allowed. So instead of having a `Diagram` case that contains other `Diagrams`, we created an extra protocol `DiagramLike`, and change our `Diagram` definition accordingly:

```
enum Diagram {
    case Prim(CGSize, Primitive)
    case Beside(DiagramLike, DiagramLike)
    case Below(DiagramLike, DiagramLike)
    case Attributed(Attribute, DiagramLike)
    case Align(Vector2D, DiagramLike)
}
```

The `DiagramLike` protocol has just one function, and only one instance:

```
protocol DiagramLike { func diagram() -> Diagram }

extension Diagram: DiagramLike {
  func diagram() -> Diagram { return self }
}
```

The `Attribute` enum is a datatype for describing different attributes of diagrams. Currently, it only supports `FillColor`, but it could easily be extended to support attributes for stroking, gradients, text attributes, etcetera:

```
enum Attribute {
    case FillColor(NSColor)
}
```

Calculating and Drawing

Calculating the size for the `Diagram` datatype is easy. The only case that's not straightforward is for `Beside` and `Below`. In case of beside, the width is equal to the sum of the widths, and the height is equal to the maximum height of the left and right diagram. For below, it's a similar pattern. For all the other cases we just call size recursively.

```
extension Diagram {
  var size : CGSize {
    switch self {
      case .Prim(let size, _):
        return size
      case .Attributed(_, let x):
        return x.diagram().size
      case .Beside(let l, let r):
        let sizeL = l.diagram().size
        let sizeR = r.diagram().size
      return CGSizeMake(sizeL.width+sizeR.width,max(sizeL.height,sizeR.height))
       case .Below(let l, let r):
```

```
        let sizeL = l.diagram().size
        let sizeR = r.diagram().size
      return CGSizeMake(max(sizeL.width,sizeR.width),sizeL.height+sizeR.height)
       case .Align(_, let r):
        return r.diagram().size
    }
  }

}
```

Before we start drawing, we will first define one more function. The `fit` function takes an alignment vector (which we used in the `Align` case of a diagram), an input size (i.e. the size of a diagram) and a rectangle that we want to fit the input size into. The input size is defined relatively to the other elements in our diagram. We scale it up, and maintain its aspect ratio.

```
func fit(alignment: Vector2D, inputSize: CGSize, rect: CGRect) -> CGRect {
    let div = rect.size / inputSize
    let scale = min(div.width, div.height)
    let size = scale * inputSize
    let space = alignment.size * (size - rect.size)
    let result = CGRect(origin: rect.origin - space.point, size: size)
    return result
}
```

For example, if we fit and center square of 1x1 into a rectangle of 200x100, we get the following result:

```
fit(Vector2D(x: 0.5, y: 0.5), CGSizeMake(1,1), CGRectMake(0,0,200,100))

> (50.0,0.0,100.0,100.0)
```

To align the rectangle to the left, we would do the following:

```
fit(Vector2D(x: 0, y: 0.5), CGSizeMake(1,1), CGRectMake(0,0,200,100))

> (0.0,0.0,100.0,100.0)
```

Now that we can represent diagrams and calculate their sizes, we're ready to draw them. We use pattern matching to make it easy to know what to draw. The `draw` method takes a couple of parameters: the context to draw in, the bounds to draw in and the actual diagram. Given the bounds, the diagram will try to fit itself into the bounds using the `fit` function defined before. For example, when we draw an ellipse, we center it and make it fill the available bounds:

```
func draw(context: CGContextRef, bounds: CGRect, diagram: Diagram) {
  switch diagram {
    case .Prim(let size, .Ellipsis):
      let frame = fit(Vector2D(x: 0.5, y: 0.5), size, bounds)
      CGContextFillEllipseInRect(context, frame)
```

For rectangles, this is almost the same, except that we call a different Core Graphics function. You might note that the `frame` calculation is the same as for ellipses. It would be possible to pull this out and have a nested switch statement, but we think is is more readable when presenting in book-form.

```
case .Prim(let size, .Rectangle):
    let frame = fit(Vector2D(x: 0.5, y: 0.5), size, bounds)
    CGContextFillRect(context, frame)
```

In the current version of our library, all text is set in the system font with a fixed size. It's very possible to make this an attribute, or change the `Text` primitive to make this configurable. In its current form though drawing text works like this:

```
case .Prim(let size, .Text(let text)):
    let frame = fit(Vector2D(x: 0.5, y: 0.5), size, bounds)
   let attributes = [NSFontAttributeName: NSFont.systemFontOfSize(12)]
 let attributedText = NSAttributedString(string: text, attributes: attributes)
    attributedText.drawInRect(frame)
```

The only attribute we support is fill color. It's very easy to add support for extra attributes, but we left that out for brevity. To draw a diagram with a `FillColor` attribute, we save the current graphics state, set the fill color, draw the diagram and finally restore the graphics state:

```
case .Attributed(.FillColor(let color), let d):
    CGContextSaveGState(context)
    color.set()
    draw(context, bounds, d.diagram())
    CGContextRestoreGState(context)
```

To draw two diagrams next to each other, we first need to find their respective frames. We created a function `splitHorizontal` that splits a `CGRect` according to a ratio (in this case, the relative size of the left diagram). Then we draw both diagrams with their frames.

```
case .Beside(let left, let right):
    let l = left.diagram()
    let r = right.diagram()
   let (lFrame, rFrame) = splitHorizontal(bounds, l.size/diagram.size)
    draw(context, lFrame, l)
    draw(context, rFrame, r)
```

The case for `Below` is exactly the same, except that we split the `CGRect` vertically instead of horizontally. This code was written to run on the Mac, and therefore the order is `bottom` and `top` (unlike UIKit, the Cocoa coordinate system has the origin at the bottom left).

```
case .Below(let top, let bottom):
    let t = top.diagram()
    let b = bottom.diagram()
    let (lFrame, rFrame) = splitVertical(bounds, b.size/diagram.size)
    draw(context, lFrame, b)
    draw(context, rFrame, t)
```

Our last case is aligning diagrams. Here, we can reuse the fit function that we defined earlier to calculate new bounds that fit the diagram exactly.

```
    case .Align(let vec, let d):
        let diagram = d.diagram()
        let frame = fit(vec, diagram.size, bounds)
        draw(context, frame, diagram)
  }
}
```

We've now defined the core of our library. All the other things can be built on top of these primitives.

Creating Views and PDFs

We can create a subclass of NSView that performs the drawing, which is very useful when working with playgrounds or when you want to draw these diagrams in Mac applications.:

```
class Draw : NSView {
    let diagram: Diagram

    init(frame frameRect: NSRect, diagram: Diagram) {
        self.diagram = diagram
        super.init(frame:frameRect)
    }

    required init(coder: NSCoder) {
        fatalError("NSCoding not supported")
    }

    override func drawRect(dirtyRect: NSRect) {
     draw(NSGraphicsContext.currentContext().cgContext, self.bounds, diagram)
    }
}
```

Now that we have an NSView, it's also very simple to make a PDF out of our diagrams. We calculate the size and just use NSViews method `dataWithPDFInsideRect` to get the PDF data.

```
func pdf(diagram: Diagram, width: CGFloat) -> NSData {
    let v : Draw = {
        let unitSize = diagram.size
        let height = width * (unitSize.height/unitSize.width)
      return Draw(frame: NSMakeRect(0, 0, width, height), diagram: diagram)
    }()
    return v.dataWithPDFInsideRect(v.bounds)
}
```

Extra combinators

To make the construction of diagrams easier, it's nice to add some extra functions (also called combinators). This is a common pattern in functional libraries: have a small set of core datatypes and functions, and then build convenience functions on top of that. For example, for rectangles, circles, text and squares we can define convenience functions:

```
func rect(#width: CGFloat, #height: CGFloat) -> Diagram {
    return Diagram.Prim(CGSizeMake(width, height), .Rectangle)
}

func circle(#radius: CGFloat) -> Diagram {
    return Diagram.Prim(CGSizeMake(radius, radius), .Ellipsis)
}

func text(#width: CGFloat, #height: CGFloat, text theText: String) -> Diagram {
    return Diagram.Prim(CGSizeMake(width, height), .Text(theText))
}

func square(#side: CGFloat) -> Diagram {
  return rect(width: side, height: side)
}
```

Also, it turns out that it's very convenient to have operators for combining diagrams horizontally and vertically, making the code more readable. They are just wrappers around `Beside` and `Below`.

```
infix operator ||| { associativity left }
func ||| (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Beside(l, r)
}

infix operator --- { associativity left }
func --- (l: Diagram, r: Diagram) -> Diagram {
    return Diagram.Below(l, r)
}
```

We can also extend the `Diagram` type and add methods for filling and alignment. Instead, we might have defined these methods as top-level functions. This is a matter of style, one is not more powerful than the other.

```
extension Diagram {
    func fill(color: NSColor) -> Diagram {
      return Diagram.Attributed(Attribute.FillColor(color), self)
    }

    func alignTop() -> Diagram {
        return Diagram.Align(Vector2D(x: 0.5,y: 1), self)
    }
```

```
    func alignBottom() -> Diagram {
        return Diagram.Align(Vector2D(x:0.5, y: 0), self)
    }
}
```

Finally, we can define an empty diagram and a way to horizontally concatenate a list of diagrams. We can just use the array's `reduce` function to do this.

```
let empty : Diagram = rect(width: 0, height: 0)

func hcat(diagrams: [Diagram]) -> Diagram {
    return diagrams.reduce(empty, combine: |||)
}
```

By adding these small helper functions, we have a powerful library for drawing diagrams. Many things are still missing but can be added easily. For example, it's straightforward to add more attributes and styling options. A bit more complicated would be adding transformations (e.g. rotation), but this too is doable.

Chapter 7

Generators and Sequences

What this chapter is about

In this chapter, we'll look at generators and sequences. These form the machinery underlying Swift's for-loops and will be the basis of our parsing library, that we will present in the following chapters.

Generators

In Objective-C and Swift, we almost always use the Array datatype to represent a list of items. It is both simple and fast. There are situations, however, where Arrays are not suitable. For example, you might not want to calculate all the elements of an the Array – because there are infinitely many or you don't expect to use them all. In such situations, you may want to use a generator instead.

We will try to provide some motivation for generators, using familiar examples from array computations. Swift's for-loops can be used to iterate over array elements:

```
for x in xs {
    \\ do something with x
}
```

In such a for-loop, the array is traversed from beginning to end. There may be examples, however, where you want to traverse arrays in a different order. This is one example where generators may be useful.

Conceptually, a generator is 'process' that generates new array elements on request. A generator is any type that adheres to the following protocol:

```
protocol GeneratorType {
    typealias Element
    func next() -> Element?
}
```

This protocol requires an associated type, `Element`, defined by the `GeneratorType`. There is a single method, `next`, that produces the next element, if it exists, and nil otherwise.

For example, the following generator produces array indices, starting from the end of an array until it reaches 0:

```
class CountdownGenerator : GeneratorType {

    typealias Element = Int

    var element : Element

    init<T>(array:[T]) {
        self.element = array.count - 1
    }

    init(start:Int) {
        self.element = start
    }

    func next() -> Element? {
        return self.element < 0 ? nil : element--
    }
}
```

For the sake of convenience we provide two initializers: one that counts down from an initial number `start`; the other is passed an array and initializes the `element` to the the array's last valid index.

We can use this `CountdownGenerator` to traverse an array backwards:

```
let xs = ["A","B","C"]
let generator = CountdownGenerator(array:xs)
while let i = generator.next() {
    println("Element \(i) of the array is \(xs[i])")
}
```

Althought it may seem like overkill on such simple examples, the generator encapsulates the computation of array indices. If we want to compute the indices in a different order, we only need to update the generator and never the code that uses it.

Generators need not produce a nil value at some point. For example, we can define a generator that produces an infinite series of powers of 2:

```
class PowerGenerator : GeneratorType {

    typealias Element = Int

    var power : Int = 1

    func next() -> Element? {
        let result = power
        power *= 2
        return result
    }
}
```

We can use the PowerGenerator to inspect increasingly large array indices, for example, when implementing a exponential search algorithm which doubles the array index in every iteration.

We may also want to use the PowerGenerator for something entirely different. Suppose we want to search through the powers of two, looking for some interesting value. The findPower function takes a predicate of type Int -> Bool as argument and returns the smallest power of two that satisfies this predicate:

```
func findPower(predicate : Int -> Bool) -> Int {
    let g = PowerGenerator()
    while let x = g.next() {
        if predicate(x) {
            return x
        }
    }
    return 0;
}
```

We can use the findPower function to compute the smallest power of two larger than 1000:

```
findPower{x in x >= 1000}
```

The generators we have seen so far all produce elements of type Int, but this need not be the case. We can equally well write generators that produce some other value. For example, the following generator produces a list of Strings, corresponding to the lines of a file.

```
class FileLinesGenerator : GeneratorType
{
    typealias Element = String

    var lines : [String]

    init(filename : String) {
        if let contents = String.stringWithContentsOfFile(filename,
                                          encoding: NSASCIIStringEncoding,
                                                             error: nil) {
            let newLine = NSCharacterSet.newlineCharacterSet()
            lines = contents.componentsSeparatedByCharactersInSet(newLine)
        } else {
            lines = []
        }
    }

    func next() -> Element? {
        if let nextLine = lines.first {
            lines.removeAtIndex(0)
            return nextLine
        } else {
```

```
            return nil
        }
    }

}
```

By defining generators in this fashion, we separate the generation of data from its usage. The generation may involve opening a file or URL and handling the errors that may arise. Hiding this behind a simple generator protocol, helps keep the code that manipulates the generated data oblivious to these issues.

By defining a protocol for generators, we can also write functions that are generic over any generator. For instance, our previous `findPower` function can be generalized as follows:

```
func find <G : GeneratorType> (var generator : G,
                               predicate : G.Element -> Bool) -> G.Element? {

    while let x = generator.next() {
        if predicate(x) {
            return x
        }
    }
    return nil
}
```

The `find` function is generic over any possible generator. The most interesting thing about it, is its type signature. The `find` function takes two arguments: a generator and a predicate. The generator may be modified by the find function, resulting from the calls to `next`, and hence we need to add the `var` attribute in the type declaration. The predicate should be a function mapping generated elements to `Bool`. We can refer to the generator's associated type as `G.Element` in the type signature of `find`. Finally, note that we may not succeed finding a value that satisfies the predicate. For that reason, `find` returns optional value, returning nil when the generator is exhausted.

It is also possible to combine generators on top of one another. For example, you may want to limit the number of items generated, buffer the generated values, or encrypt the data generated somehow. Here is one simple example of a generator transformer that produces at most `limit` values from its argument generator:

```
class LimitGenerator<G : GeneratorType> : GeneratorType
{
    typealias Element = G.Element
    var limit = 0
    var generator : G

    init (limit : Int, generator : G) {
        self.limit = limit
        self.generator = generator
    }

    func next() -> Element? {
```

```
        if limit >= 0 {
            limit--
            generator.next()
        }
        else {
            return nil
        }
    }
}
```

Such a generator may be useful to populate an array of fixed size or buffer the elements generated somehow.

When writing generators, it can sometimes be cumbersome to introduce new classes for every generator. Swift provides a simple struct, `GeneratorOf<T>`, that is generic in the element type. It can be initialized with a `next` function:

```
struct GeneratorOf<T> : GeneratorType, SequenceType {
    init(next: () -> T?)
    ...
```

We will provide the complete definition of `GeneratorOf` shortly. For now, we'd like to point out that the `GeneratorOf` struct not only implements the `GeneratorType` protocol, but also implements the `SequenceType` protocol that we will cover in the next section.

Using `GeneratorOf` allows for much shorter definitions of generators. For example, we can rewrite our `CountdownGenerator` as follows:

```
func countDown(start:Int) -> GeneratorOf<Int> {
    var i = start
    return GeneratorOf {return i < 0 ? nil : i--}
}
```

We can even define functions to manipulate and combine generators in terms of `GeneratorOf`. For example, we can append two generators with the same underlying element type as follows:

```
func +<A>(var first: GeneratorOf<A>, var second: GeneratorOf<A>) -> GeneratorOf<A> {
    return GeneratorOf {
        if let x = first.next() {
            return x
        } else if let x = second.next() {
            return x
        }
        return nil
    }
}
```

The resulting generator simply reads off new elements from its `first` argument generator; once this is exhausted, it produces elements from its `second` generator. Once both generators have returned nil, the composite generator also returns nil.

Sequences

Generators form the basis of another Swift protocol, sequences. Generators provide a 'one-shot' mechanism for repeatedly computing a next element. There is no way to rewind or replay the elements generated. The only thing we can do is create a fresh generator and use that instead. The SequenceType protocol provides just the right interface for doing that:

```
protocol SequenceType {
    typealias Generator : GeneratorType
    func generate() -> Generator
}
```

Every sequence has an associated generator type and a method to create a new generator. We can then use this generator to traverse the sequence. For example, we can use our `CountdownGenerator` to define a sequence that generates a series of array indexes in back-to-front order:

```
struct ReverseSequence<T> : SequenceType {
    var array : [T]

    init (array : [T]) {
        self.array = array
    }

    typealias Generator = CountdownGenerator
    func generate() -> Generator {
        return CountdownGenerator(array:array)
    }
}
```

Every time we want to traverse the array stored in the `ReverseSequence` struct, we can call the `generate` method to produce the desired generator. The following example shows how to fit these pieces together:

```
let reverseSequence = ReverseSequence(array:xs)
let reverseGenerator = reverseSequence.generate()
while let i = reverseGenerator.next() {
  print("Index \(i) is \(xs[i])")
}
```

In contrast to the previous example that just used the generator, the same sequence can be traversed a second time – we would simply call `generate` to produce a new generator.

Swift has special syntax for working with sequences. Instead of creating the generator associated with a sequence yourself, you can write a for-in loop. For example, we can also write the previous code snippet as:

```
for i in ReverseSequence(array:xs) {
  print("Index \(i) is \(xs[i])")
}
```

Under the hood, Swift then uses the `generate` method to produce a generator and repeatedly call its `next` function until it produces nil.

The obvious drawback of our `CountdownGenerator` is that it produces numbers, while we may be interested in the elements associated with an array. Fortunately there are standard `map` and `filter` functions that manipulate sequences rather than arrays:

```
func filter<S : SequenceType>
  (source: S, includeElement: (S.Generator.Element) -> Bool) -> [S.Generator.Element]

func map<S : SequenceType, T>
  (source: S, transform: (S.Generator.Element) -> T) -> [T]
```

To produce the elements of an array in reverse order, we can `map` over our `ReverseSequence`:

```
let reverseElements = map(ReverseSequence(array:xs)){i in xs[i]}
for x in reverseElements {
  print("Element is \(x)")
}
```

Similarly, we may of course want to filter out certain elements from a sequence.

It is worth pointing out that these `map` and `filter` functions do not return new sequences, but traverse the sequence to produce an array. Mathematicians may therefore object to calling such an operation a `map` as it fails to leave the underlying structure (a sequence) intact. There are separate versions of `map` and `filter` that do produce sequences. These are defined as extensions of the `LazySequence` class. A `LazySequence` is simple wrapper around regular sequences:

```
func lazy<S : SequenceType>(s: S) -> LazySequence<S>
```

If you need to map or filter sequences that may produce infinite results or many results that you may not be interested in, be sure to use a `LazySequence` rather than a `Sequence`. Failing to do so could cause your program to diverge or take much longer than you might expect.

Case study: Better shrinking in QuickCheck

In this section we will give a somewhat larger case study of defining sequences by improving the `Smaller` protocol we implemented in the QuickCheck chapter. Originally, the protocol was defined as follows:

```
protocol Smaller {
    func smaller() -> Self?
}
```

We used the `Smaller` protocol to try and shrink counterexamples that our testing uncovered. The `smaller` function was repeatedly called to generate a smaller value, if this value also fails the test it was considered a 'better' counterexample than the original one. The `Smaller` instance we defined for arrays simply tried to repeatedly strip off the first element:

```
extension Array : Smaller {
    func smaller() -> [T]? {
      return self.isEmpty ? nil : Array(self[startIndex.successor()..<endIndex])
    }
}
```

While this will certainly help shrink counterexamples in some examples, there are many different ways to shrink an array. Computing all possible sub-arrays is quite an expensive operation. For an array of length n there are 2^n possible sub-arrays that may or may not be interesting counterexamples: generating and testing them is not a good idea.

Instead, we will show how to use a generator to produce a series of smaller values. We can then adapt our QuickCheck library to use the following protocol:

```
protocol Smaller {
    func smaller() -> GeneratorOf<Self>
}
```

When QuickCheck finds a counterexample, we can then rerun our tests on the series of smaller values, until we have found a suitably small counterexample. The only thing we still have to do, is write a `smaller` function for arrays (and any other type that we might want to shrink).

As a first step, instead of removing just the first element of the array, we will compute a series of arrays, where each new array has one element removed. This will not produce all possible sublists, but only a sequence of arrays that are all one element shorter than the original array. Using `GeneratorOf`, we can define such a function as follows:

```
func removeAnElement<T>(var array: [T]) -> GeneratorOf<[T]> {
    var i = 0
    return GeneratorOf {
        if i < array.count {
            var result = array.self
            result.removeAtIndex(i)
            i++
            return result
        }
        return nil
    }
}
```

The `removeAnElement` function keeps track of a variable `i`. When asked for a next element, it checks whether or not `i` is less than the length of the array. If so, it computes a new array, `result`, and increments `i`. If we have reached the end of our original array, we return nil. Note that we duplicate the array using `array.self` to ensure that elements that are removed in one iteration are still present in the next.

We can now see that this returns all possible arrays that are one element smaller:

```
removeAnElement([1,2,3])
```

Unfortunately, this call does not produce the desired result – it defines a `GeneratorOf<[Int]>`, while we would like to see an array of arrays. Fortunately, there is an `Array` initializer that takes a `Sequence` as argument. Using that initializer, we can test our generator as follows:

```
Array(removeAnElement([1,2,3]))
```

A more functional approach

Before we refine the `removeElement` function further, we will rewrite it in a more functional way. The implementation of `removeElement` we gave above uses quite some explicit copying of arrays and mutable state. We have already seen that working with data types and recursion forms a powerful technique for decomposing problems into smaller pieces. While the Array type is not a data type, we can define a pattern matching principle on it ourselves:

```
extension Array {
    var match : (head: T, tail: [T])? {
      return (count > 0) ? (self[0],Array(self[1..<count])) : nil
    }
}
```

In case of an empty array, `match` returns `nil`; when the array is non-empty array, it returns a tuple with the first element and the rest of the array. We can use this to define recursive traversals of arrays. For example, we can sum the elements of an array recursively, without using a for-loop or reduce, as follows:

```
func sum(xs : [Int]) -> Int {
    if let (head,tail) = xs.match
    {
        return (head + sum(tail))
    } else {
        return 0
    }
}
```

Of course, this may not be a very good example: a function like `sum` is easy to write using `reduce`. This is not true for all functions on arrays. For example, consider the problem of inserting a new element into a sorted array. Writing this using `reduce` is not at all easy; writing it as a recursive function is fairly straightforward:

```
func insert(x : Int, xs : [Int]) -> [Int] {
    if let (head,tail) = xs.match
    {
        return (x <= head ? [x] + xs : [head] + insert(x,tail))
    } else {
        return [x]
    }
}
```

Before we can return to our original problem, how to shrink an array, we need one last auxiliary definition. In the Swift standard library, there is a `GeneratorOfOne` struct that can be useful for wrapping an optional value as a generator:

```
struct GeneratorOfOne<T> : GeneratorType, SequenceType {
    init(_ element: T?)
    ...
```

Given an optional element, it generates the sequence with just that element (provided it is non-nil):

```
let three : [Int] = Array(GeneratorOfOne(3))
let empty : [Int] = Array(GeneratorOfOne(nil))
```

For the sake of convenience, we will define our own little wrapper function around `GeneratorOfOne`:

```
func one<X>(x: X?) -> GeneratorOf<X> {
    return GeneratorOf(GeneratorOfOne(x))
}
```

Now we finally return to our original problem, redefining the `smaller` function on arrays. If we try to formulate a recursive pseudocode definition of what our original `removeElement` function computed we might arrive at something along the following lines:

- If the array is empty, return nil;
- If the array can be split into a head and tail, we can recursively compute the remaining sub-arrays as follows:
  - tail of the array is a sub-array
  - if we prepend `head` to all the sub-arrays of the tail, we can compute the sub-arrays of the original array.

We can translate this algorithm directly into Swift with the functions we have defined:

```
func smaller1<T>(array: [T]) -> GeneratorOf<[T]> {
    if let (head,tail) = array.match {
        let gen1 : GeneratorOf<[T]> = one(tail)
     let gen2 : GeneratorOf<[T]> = map(smaller1(tail),{smallerTail in [head] + smallerTail}
        return gen1 + gen2
    } else {
        return one(nil)
    }
}
```

We're now ready to test our functional variant, and we can verify that it's the same result as `removeAnElement`:

```
Array(smaller1([1,2,3]))
```

Note that there is one thing we should point out. In this definition of `smaller` we are using our own version of `map`:

```
func map<A,B>(var g: GeneratorOf<A>, f: A -> B) -> GeneratorOf<B> {
    return GeneratorOf {
        g.next().map(f)
    }
}
```

You may recall that the `map` and `filter` methods from the standard library return a `LazySequence`. To avoid the overhead of wrapping and unwrapping these lazy sequences, we have chosen to manipulate the `GeneratorOf` directly.

There is one last improvement worth making. There is one more way to try and reduce the counterexamples that QuickCheck finds. Instead of just removing elements, we may also want to try and shrink the elements themselves. To do that, we need to add a condition that T conforms to the smaller protocol.

```
func smaller<T : Smaller>(ls: [T]) -> GeneratorOf<[T]> {
  if let (head, tail) = ls.match {
        let gen1 : GeneratorOf<[T]> = one(tail)
       let gen2 : GeneratorOf<[T]> = map(smaller(tail), {xs in [head] + xs})
        let gen3 : GeneratorOf<[T]> = map(head.smaller(), {x in [x] + tail})
         return gen1 + gen2 + gen3
  } else {
    return one(nil)
  }
}
```

We can check the results of our new `smaller` function.

```
Array(smaller([1,2,3]))
```

Besides generating sublists, this new version of the `smaller` function also produces arrays where the values of the elements is smaller.

Beyond map and filter

In the coming chapter we will need a few more operations on sequences and generators. We have already defined a concatenation, +, on generators. Can we use this definition to concatenate sequences?

We might try to reuse our + operation on generators when concatenating sequences as follows:

```
func +<A>(l: SequenceOf<A>, r: SequenceOf<A>) -> SequenceOf<A> {
  return SequenceOf(l.generate() + r.generate())
}
```

This definition calls the generate method of the two argument sequences, concatenates these, and assigns the resulting generator to the sequence. Unfortunately, it does not quite work as expected. Consider the following example:

```
let s = SequenceOf([1,2,3]) + SequenceOf([4,5,6])
println("First pass:")
for x in s {
    print(x)
}
println("\nSecond pass:")
for x in s {
    print(x)
}
```

We construct a sequence containing the elements `[1,2,3,4,5,6]` and traverse it twice, printing the elements we encounter. Somewhat suprisingly perhaps, this code produces the following output:

```
First pass:
123456
Second pass:
```

The second for loop is not producing any output – what went wrong? The problem is in the definition of concatenation on sequences. We assemble the desired generator, `l.generate() + r.generate()`. This generator produces all the desired elements in the first loop in the example above. Once it has been exhausted, however, traversing the compound sequence a second time will not produce a fresh generator, but instead use the generator that has already been exhausted.

Fortunately, this problem is easy to fix. We need to ensure that the result of our concatenation operation can produce new generators. To do so, we pass a function that produces generators, rather than a fixed generator to the `SequenceOf` initializer:

```
func +<A>(l: SequenceOf<A>, r: SequenceOf<A>) -> SequenceOf<A> {
  return SequenceOf { l.generate() + r.generate() }
}
```

Now, we can iterate over the same sequence multiple times. When writing your own methods that combine sequences, it is important to ensure that every call to `generate()` produces a fresh generator that is oblivious to any previous traversals.

So far we can concatenate two sequences. What about flattening a sequence of sequences? Before we deal with sequences, let's try writing a `join` operation on generators that, given a `GeneratorOf<GeneratorOf<A>>`, produces a `GeneratorOf<A>`:

```
struct JoinedGenerator<A> : GeneratorType {
    typealias Element = A

    var generator: GeneratorOf<GeneratorOf<A>>
    var current: GeneratorOf<A>?

    init(_ g: GeneratorOf<GeneratorOf<A>>) {
        generator = g
        current = generator.next()
    }

    mutating func next() -> A? {
        if var c = current {
            if let x = c.next() {
                return x
            } else {
                current = generator.next()
                return next()
            }
        }
        return nil
    }
}
```

This `JoinedGenerator` maintains two pieces of mutable state: an optional `current` generator and the remaining `generators`. When asked to produce the next element, it calls the `next` function on the current generator, if it exists. When this fails, it updates the `current` generator and recursively calls `next` again. Only when all the generators have been exhausted, does the `next` function return nil.

Next, we use this `JoinedGenerator` to join a sequence of sequences:

```
func join<A>(s: SequenceOf<SequenceOf<A>>) -> SequenceOf<A> {
   return SequenceOf {JoinedGenerator(map(s.generate()) {g in g.generate()})}
}
```

The argument of `JoinedGenerator` may look complicated, but it does very little. When struggling to understand an expression like this, following the types is usually a good way to learn what it does. We need to provide an argument closure producing a value of type `GeneratorOf<GeneratorOf<A>>`; calling `s.generate()` gets us part of the way there, producing a value of type `GeneratorOf<SequenceOf<A>>`. The only thing we need to do is call `generate` on all the sequences inside the resulting generators, which is precisely what the call to `map` accomplishes.

Finally, we can also combine `join` and `map` to write the following `flatmap` function:

```
func flatmap<A,B>(xs: SequenceOf<A>, f: A -> SequenceOf<B>) -> SequenceOf<B> {
    return join(map(xs,f))
}
```

Given a sequence of A elements, and a function `f` that, given a single value of type A, produces a new sequence of B elements, we can build a single sequence of B elements. To do so, we simply map `f` over the argument sequence, constructing a `SequenceOf<SequenceOf<B>>`, which we `join` to obtain the desired `SequenceOf<B>`.

Now, we've got a good grip on sequences and the operations they support, we can start to write our parser combinator library.

Chapter 8

Parser Combinators

Note: this chapter isn't copy-edited yet, so there's no need to file issues for things like spelling mistakes. But please do file issues if you find anything unclear or any other kind of feedback regarding the content.

Parsers are a very useful tool: they take a list of tokens (usually, a list of characters) and transform it into a structure. Often, parsers are generated using an external tool, such as Bison or YACC. Instead of using an external tool, we'll build a parser library in this chapter, which we can use later for building our own parser. Functional languages are very well suited for this task.

There are several approaches to writing a parsing library. Here we'll build a parser combinator library. A parser combinator is a higher order function that takes several parsers as input and returns a new parser as its output. The library we'll build is an almost direct port of a Haskell library [1], with a few modifications.

We will start with defining a couple of core combinators. On top of that, we will build some extra convenience functions, and finally, we will show an example that parses arithmetic expressions, such as `1 + 3 * 3`, and calculates the result.

The Core

In this library, we'll make heavy use of sequences and slices.

We define a parser as a function that takes a slice of tokens, processes some of these tokens, and returns a tuple of the result and the remainder of the tokens. To make our lifes a bit easier, we wrap this function in a struct (otherwise we'd have to write out the entire type every time). We make our parser generic over 2 types: Token and Result.

```
struct Parser<Token, Result> {
    let p: Slice<Token> -> SequenceOf<(Result, Slice<Token>)>
}
```

We'd rather use a type alias to define our parser type, but type aliases don't support generic types. Therefore we have to live with the indirection of using a struct in this case.

[1] The code presented here is directly translated from Haskell into Swift. S. Doaitse Swierstra, Combinator Parsing: A Short Tutorial. http://www.cs.tufts.edu/~nr/cs257/archive/doaitse-swierstra/combinator-parsing-tutorial.pdf

Let's start with a very simple parser that parses the single character "a". To do this we write a function

```
func parseA() -> Parser<Character, Character>
```

that returns the "a" character parser. Note that it returns a parser with the token type `Character` as well as the result type `Character`. The results of this parser will be tuples of an "a" character and the remainder of characters. It works like this: it splits the input stream into head (the first character) and tail (all remaining characters), and returns a single result if the first character is an "a". If the first character isn't an "a", the parser fails by returning `none()`, which is simply an empty sequence.

```
func parseA() -> Parser<Character, Character> {
    let a : Character = "a"
    return Parser { x in
        if let (head, tail) = x.match {
            if head == a {
                return one((a, tail))
            }
        }
        return none()
    }
}
```

We can test it using the `testParser` function. This runs the parser given by the first argument over the input string that is given by the second argument. The parser will generate a sequence of possible results, which get printed out by the `testParser` function. Usually, we are only interested in the very first result.

```
testParser(parseA(), "abcd")
```

```
> Success, found a, remainder: [b, c, d]
```

If we run the parser on a string that doesn't contain an "a" at the start we get a failure:

```
testParser(parseA(), "test")
```

```
> Parsing failed.
```

We can easily abstract this function to work on any character. We pass in the character as a parameter, and only return a result if the first character in the stream is the same as the parameter.

```
func parseCharacter(character: Character) -> Parser<Character, Character> {
    return Parser { x in
        if let (head, tail) = x.match {
            if head == character {
                return one((character, tail))
            }
        }
        return none()
    }
}
```

Now we can test our new method:

```
testParser(parseCharacter("t"), "test")

> Success, found t, remainder: [e, s, t]
```

We can abstract this method one final time, making it generic over any kind of token. Instead of checking if the token is equal, we pass in a function with type `Token -> Bool`, and if the function returns true for the first character in the stream, we return it.

```
func satisfy<Token>(condition: Token -> Bool) -> Parser<Token, Token> {
    return Parser { x in
        if let (head, tail) = x.match {
            if condition(head) {
                return one((head, tail))
            }
        }
        return none()
    }
}
```

Now we can define a function `symbol` that works like `parseCharacter`, with the only difference that it can be used with any type that conforms to `Equatable`.

```
func symbol<Token: Equatable>(symbol: Token) -> Parser<Token, Token> {
    return satisfy { $0 == symbol }
}
```

Choice

Parsing a single symbol isn't very useful, unless we add functions to combine two parsers. The first function that we will introduce is the choice operator, and it can parse using either the left operand or the right operand. It is implemented in a simple way: given an input string, it runs the left operand's parser, which yields a sequence of possible results. Then it runs the right operand, which also yields a sequence of possible results, and it concatenates the two sequences. Note that the left and the right sequences might both be empty, or contain a lot of elements. Because they are calculated lazily, it doesn't really matter.

```
infix operator <|> { associativity right precedence 130 }
func <|> <Token, A>(l: Parser<Token, A>, r: Parser<Token, A>) -> Parser<Token, A> {
    return Parser { input in
        l.p(input) + r.p(input)
    }
}
```

To test our new operator we build a parser that parses either an a or a b:

```
let a: Character = "a"
let b: Character = "b"

testParser(symbol(a) <|> symbol(b), "bcd")

> Success, found b, remainder: [c, d]
```

Sequence

To combine two parsers that happen after each other we'll start with a more naive approach and expand that later to something more convenient and powerful. First we write a `sequence` function:

```
func sequence<Token, A, B>(l: Parser<Token, A>,
                           r: Parser<Token, B>) -> Parser<Token, (A,B)>
```

The returned parser first uses the left parser to parse something of type A. Let's say we wanted to parse the string "xyz" for an "x" immediately followed by a "y". The left parser (the one looking for an "x") would then generate the following sequence containing a single (result, remainder) tuple:

```
[ ("x", "yz") ]
```

Applying the right parser to the remainder ("yz") of the left parser's tuple yields another sequence with one tuple:

```
[ ("y", "z") ]
```

We then combine those tuples by grouping the "x" and "y" into a new tuple ("x", "y"):

```
[ (("x", "y"), "z") ]
```

Since we are doing these steps for each tuple in the returned sequence of the left parser, we end up with a sequence of sequences:

```
[ [ (("x", "y"), "z") ] ]
```

Finally, we flatten this structure to a simple sequence of `((A, B), Slice<Token>)` tuples. In code, the whole `sequence` function looks like this:

```
func sequence<Token, A, B>(l: Parser<Token, A>,
                           r: Parser<Token, B>) -> Parser<Token, (A,B)> {

    return Parser { input in
        let leftResults = l.p(input)
        return flatMap(leftResults) { a, leftRest in
            let rightResults = r.p(leftRest)
```

```
            return map(rightResults, { b, rightRest in
                ((a, b), rightRest)
            })
        }
    }
}
```

Note that the above parser only succeeds if both `l` and `r` succeed: if they don't, no tokens are consumed.

We can test our parser by trying to parse a sequence of an "x" followed by a "y":

```
let x: Character = "x"
let y: Character = "y"

let p : Parser<Character, (Character, Character)> = sequence(symbol(x), symbol(y))
testParser(p, "xyz")

> Success, found (x, y), remainder: [z]
```

Refining sequences

It turns out that the `sequence` function we wrote above is a naive approach to combine multiple parsers that are applied after each other. Imagine we wanted to parse the same string "xyz" as above, but this time we want to parse "x" followed by "y" followed by "z". We could try to use the `sequence` function in a nested way to combine three parsers:

```
let z: Character = "z"

let p2 = sequence(sequence(symbol(x), symbol(y)), symbol(z))
testParser(p2, "xyz")

> Success, found ((x, y), z), remainder: []
```

The problem of this approach is that it yields a nested tuple `(("x", "y"), "z")` instead of a flat one `("x", "y", "z")`. To rectify this we could write a `sequence3` function that combines three parsers instead of just two:

```
func sequence3<Token, A, B, C>(p1: Parser<Token, A>,
                               p2: Parser<Token, B>,
                          p3: Parser<Token, C>) -> Parser<Token, (A, B, C)> {

    return Parser { input in
        let p1Results = p1.p(input)
        return flatMap(p1Results) { a, p1Rest in
            let p2Results = p2.p(p1Rest)
            return flatMap(p2Results) {b, p2Rest in
                let p3Results = p3.p(p2Rest)
```

```
                return map(p3Results, { c, p3Rest in
                    ((a, b, c), p3Rest)
                })
            }
        }
    }
}

let p3 = sequence3(symbol(x), symbol(y), symbol(z))
testParser(p3, "xyz")

> Success, found (x, y, z), remainder: []
```

This returns the expected result, but this approach is way too inflexible and doesn't scale. It turns out there is a much more convenient way to combine multiple parsers in sequence.

As a first step, we create a parser that consumes no tokens at all at returns a function `A -> B`. This function takes on the job of transforming the result of one or more other parsers in the way we want it to. A very simple example of such a parser could be:

```
func integerParser<Token>() -> Parser<Token, Character -> Int> {
    return Parser { input in
        return one(({ x in String(x).toInt()! }, input))
    }
}
```

This parser doesn't consume any tokens and returns a function that takes a character and turns it into an integer. Let's use the extremely simple input stream "3" as example. Applying the `integerParser` to this input yields the sequence:

```
[ (A -> B, "3") ]
```

Applying another parser to parse the symbol "3" in the remainder (which is equal to the original input since the `integerParser` didn't consume any tokens) yields:

```
[ ("3", "") ]
```

Now we just have to create a function that combines these two parsers and returns a new parser, so that the function yielded by `integerParser` gets applied to the character "3" yielded by the symbol parser. This function looks very similar to the `sequence` function – it calls `flatMap` on the sequence returned by the first parser and then maps over the sequence returned by the second parser applied to the remainder.

The key difference is that the inner closure does not return the results of both parsers in a tuple as `sequence` did, but it applies the function yielded by the first parser to the result of the second parser:

```
func combinator<Token, A, B>(l: Parser<Token, A -> B>,
                             r: Parser<Token, A>) -> Parser<Token, B> {
```

```
    return Parser { input in
        let leftResults = l.p(input)
        return flatMap(leftResults) { f, leftRemainder in
            let rightResults = r.p(leftRemainder)
        return map(rightResults) { x, rightRemainder in (f(x), rightRemainder) }
        }
    }
}
```

Putting all of this together:

```
let three: Character = "3"

testParser(combinator(integerParser(), symbol(three)), "3")

> Success, found 3, remainder: []
```

Now we've laid the groundwork to build a really elegant parser combination mechanism.

The first thing we'll do is to refactor our `integerParser` function into a generic function with one parameter that returns a parser that always succeeds, consumes no tokens and returns the parameter we passed into the function as result:

```
func pure<Token, A>(value: A) -> Parser<Token, A> {
    return Parser { one((value, $0)) }
}
```

With this in place we can rewrite the previous example like this:

```
func toInteger(c: Character) -> Int {
    return String(c).toInt()!
}
testParser(combinator(pure(toInteger), symbol(three)), "3")

> Success, found 3, remainder: []
```

The whole trick to leverage this mechanism to combine multiple parsers lies in the concept of currying. Returning a curried function from the first parser enables us to go multiple times through the combination process, depending on the number of arguments of the curried function. For example:

```
func toInteger2(c1: Character)(c2: Character) -> Int {
    let combined = String(c1) + String(c2)
    return combined.toInt()!
}

testParser(combinator(combinator(pure(toInteger2), symbol(three)), symbol(three)), "33")

> Success, found 33, remainder: []
```

Since nesting a lot of `combinator` calls within each other is not very readable, we define an operator for it:

```
infix operator <*> { associativity left precedence 150 }
func <*><Token, A, B>(l: Parser<Token, A -> B>,
                      r: Parser<Token, A>) -> Parser<Token, B> {
    return Parser { input in
        let leftResults = l.p(input)
        return flatMap(leftResults) { f, leftRemainder in
            let rightResults = r.p(leftRemainder)
            return map(rightResults) { x, y in (f(x), y) }
        }
    }
}
```

Now we can express the previous example as:

```
testParser(pure(toInteger2) <*> symbol(three) <*> symbol(three), "33")
```

```
> Success, found 33, remainder: []
```

Notice that we have defined the `<*>` operator to have left precedence. This means that the operator will be first applied to the left two parsers, and then to the result of this operation and the right parser. In other words, this behavior is exactly the same as our nested `combinator` function calls above.

Another example how we can now use this operator is to create a parser that combines several characters into a string:

```
let aOrB = symbol(a) <|> symbol(b)
func combine(a: Character)(b: Character)(c: Character) -> String {
  return a + b + c
}
let parser = pure(combine) <*> aOrB <*> aOrB <*> symbol(b)
testParser(parser, "abb")
```

```
> Success, found abb, remainder: []
```

In chapter TODO we defined the `curry` function, which curries a function with two parameters. We can define multiple variants of curry which work on functions with different numbers of parameters. For example, we could define a variant that works on a function with three arguments:

```
func curry<A,B,C,D>(f: (A, B, C) -> D) -> A -> B -> C -> D {
  return { a in { b in { c in f(a,b,c) } } }
}
```

Now, we can write the above parser in an even shorter way:

```
let parser2 = pure(curry {$0 + $1 + $2}) <*> aOrB <*> aOrB <*> symbol(b)
testParser(parser2, "abb")
```

Convenience Combinators

Using the above combinators we can already parse a lot of interesting languages. However, they can be a bit tedious to express. Luckily, there are some extra functions we can define to make life easier. First we will define a function to parse a character from an `NSCharacterSet`. This can be used, for example, to create a parser that parses decimal digits:

```
func characterFromSet(set: NSCharacterSet) -> Parser<Character,Character> {
    return satisfy { return member(set, $0) }
}

let decimalDigit = characterFromSet(NSCharacterSet.decimalDigitCharacterSet())
```

To verify that our `decimalDigit` parser works, we can run it on an example input string:

```
testParser(decimalDigit, "012")

> Success, found 0, remainder: [1, 2]
```

The next convenience combinator we want to write is a `many` function, which executes a parser zero or more times.

```
func many<Token,A>(p: Parser<Token,A>) -> Parser<Token,[A]> {
    return (pure(prepend) <*> p <*> many(p)) <|> pure([])
}
```

The `prepend` function combines a value of type A and an array `[A]` into a new array.

However, if we would try to use this function, we will get stuck in an infinite loop. That's because of the recursive call of `many` in the return statement.

Luckily, we can use auto-closures to defer the evaluation of the recursive call to `many` until it is really needed and with that break the infinite recursion. To do that, we will first define a helper function `recurse`: it returns a parser that will only be executed once it's actually needed, because we use the `@autoclosure` keyword for the function parameter.

```
func recurse<Token, A>(f: @autoclosure () -> Parser<Token, A>) -> Parser<Token, A> {
    return Parser { f().p($0) }
}
```

Now we wrap the recursive call to `many` with this function:

```
func many<Token,A>(p: Parser<Token,A>) -> Parser<Token,[A]> {
    return (pure(prepend) <*> p <*> recurse(many(p))) <|> pure([])
}
```

Let's test the many combinator to see if it yields multiple results. As we will see later on in this chapter, we usually only use the first successful result of a parser, and the other ones will never get computed since they are lazily evaluated.

```
testParser(many(decimalDigit), "12345")

> Success, found [1, 2, 3, 4, 5], remainder: []
> Success, found [1, 2, 3, 4], remainder: [5]
> Success, found [1, 2, 3], remainder: [4, 5]
> Success, found [1, 2], remainder: [3, 4, 5]
> Success, found [1], remainder: [2, 3, 4, 5]
> Success, found [], remainder: [1, 2, 3, 4, 5]
```

Another useful combinator is `many1`, which parses something one or more times. It is defined using the `many` combinator.

```
func many1<Token, A>(p: Parser<Token, A>) -> Parser<Token, [A]> {
    return pure(prepend) <*> p <*> many(p)
}
```

If we parse one or more digits, we get back an array of digits in the form of `Characters`. To convert this into an integer, we can first convert the array of `Characters` into a string, and then just call the built-in `toInt()` function on it. Even though `toInt` might return nil, we know that it will succeed, so we can force it with the `!` operator.

```
let number = pure({ characters in string(characters).toInt()! }) <*> many1(decimalDigit)

testParser(number, "205")

> Success, found 205, remainder: []
> Success, found 20, remainder: [5]
> Success, found 2, remainder: [0, 5]
```

If we look at the code we've written so far we see one recurring pattern: `pure(x) <*> y`. In fact, it is so common that it's useful to define an extra operator for it. If we look at the type, we can see that it's very similar to a `map` function: it takes a function of type `A -> B` and a parser of type A, and returns a parser of type B.

```
func </> <Token, A, B>(l: A -> B, r: Parser<Token, A>) -> Parser<Token, B> {
    return pure(l) <*> r
}
```

Now we have defined a lot of useful functions, so it's time to start combining some of them into real parsers. For example, if we want to create a parser that can add two integers, we can now write it in the following way:

```
let plus: Character = "+"
func add(x: Int)(_: Character)(y: Int) -> Int {
  return x + y
}
let parseAddition = add </> number <*> symbol(plus) <*> number
```

And we can again verify that it works:

```
testParser(parseAddition, "41+1")
```

```
> Success, found 42, remainder: []
```

It is often the case that we want to parse something but ignore the result, for example, with the plus symbol in the parser above. We want to know that it's there, but we do not care about the result of the parser. We can define another operator, `<*`, which works exactly like the `<*>` operator, except that it throws away the right-hand result after parsing it (that's why the right angular bracket is missing in the operator name). Similarly, we will also define a `*>` operator that throws away the left-hand result:

```
func <* <Token, A, B>(p: Parser<Token, A>, q: Parser<Token, B>) -> Parser<Token, A> {
    return {x in {_ in x} } </> p <*> q
}
func *> <Token, A, B>(p: Parser<Token, A>, q: Parser<Token, B>) -> Parser<Token, B> {
    return {_ in {y in y} } </> p <*> q
}
```

Now, we can write another parser, for multiplication. It's very similar to the `parseAddition` function, except that it uses our new `<*` operator to throw away the `"*"` after parsing it.

```
let multiply : Character = "*"
let parseMultiplication = curry(*) </> number <* symbol(multiply) <*> number
testParser(parseMultiplication, "8*8")
```

```
> Success, found 64, remainder: []
```

A simple calculator

We can extend our example to parse expressions like `10+4*3`. Here, it is important to realize that when calculating the result, multiplication takes precedence over addition. This is because of a rule in mathematics (and programming) that's called order of operations. Expressing this in our parser is quite natural. Let's start with the atoms, which take the highest precedence:

```
typealias Calculator = Parser<Character, Int>

func operator0(character: Character,
               evaluate: (Int, Int) -> Int,
               next: Calculator) -> Calculator {

  return { x in { y in evaluate(x,y) } } </> next <* symbol(character) <*> next
}

func pAtom0() -> Calculator { return number }
func pMultiply0() -> Calculator { return operator0("*", *, pAtom0()) }
```

```
func pAdd0() -> Calculator { return operator0("+", +, pMultiply0()) }
func pExpression0() -> Calculator { return pAdd0() }

testParser(pExpression0(), "1+3*3")

> Parsing failed.
```

Why did the parsing fail?

First, an add expression is parsed. An add expression consists of a multiplication expression, followed by a "+" and then another multiplication expression. 3*3 is a multiplication expression, however, 1 is not. It's just a number. To fix this, we can change our `operator` function to either parse an expression of the form `next operator next`, or without the operator (just `next`):

```
func operator1(character: Character,
               evaluate: (Int, Int) -> Int,
               next: Calculator) -> Calculator {

  let withOperator = { x in { y in evaluate(x,y) } } </> next <* symbol(character) <*> next
   return withOperator <|> next
}
```

Now, we finally have a working variant:

```
func pAtom1() -> Calculator { return number }
func pMultiply1() -> Calculator { return operator1("*", *, pAtom1()) }
func pAdd1() -> Calculator { return operator1("+", +, pMultiply1()) }
func pExpression1() -> Calculator { return pAdd1() }

testParser(pExpression1(), "1+3*3")

> Success, found 10, remainder: []
> Success, found 4, remainder: [*, 3]
> Success, found 1, remainder: [+, 3, *, 3]
```

If we want to add some more operators and abstract this a bit further, we can create an array of operator characters and their interpretation functions, and use the `reduce` function to combine them into one parser:

```
typealias Op = (Character, (Int, Int) -> Int)
let operatorTable : [Op] = [("*", *), ("/", /), ("+", +), ("-", -)]

func pExpression2() -> Calculator {
    return operatorTable.reduce(number, { (next: Calculator, op: Op) in
        operator1(op.0, op.1, next)
    })
}
testParser(pExpression2(), "1+3*3")
```

```
> Success, found 10, remainder: []
> Success, found 4, remainder: [*, 3]
> Success, found 1, remainder: [+, 3, *, 3]
```

However, our parser becomes notably slow as we add more and more operators. This is because the parser is constantly backtracking: it tries to parse something, then fails, and tries another alternative. For example, when trying to parse “1+3*3“, first, the”-” operator is tried (which consists of a “+” expression, followed by a “-” character, and then another “+” expression). The first “+” expression succeeds, but because no “-” character is found, it tries the alternative: just a “+” expression. If we continue this, we can see that a lot of unnecessary work is being done.

Writing a parser like above is very simple. However, it is not very efficient. If we take a step back, and look at the grammar we’ve defined using our parser combinators, we could write it down like this (in a pseudo-grammar description language):

```
expression = min
min = add "-" add | add
add = div "+" div | div
div = mul "/" mul | mul
mul = num "*" num | num
```

To remove a lot of the duplication we can refactor this grammar like this:

```
expression = min
min = add ("-" add)?
add = div ("+" div)?
div = mul ("/" mul)?
mul = num ("*" num)?
```

Before we define the new operator function, we first define an additional variant of the `</>` operator that consumes but doesn’t use its right operand:

```
infix operator </  { precedence 170 }
func </ <Token,A,B>(l: A, r: Parser<Token,B>) -> Parser<Token,A> {
    return pure(l) <* r
}
```

Also, we will define a function `optionallyFollowed` which parses its left operand, optionally followed by another part:

```
func optionallyFollowed<A>(l: Parser<Character, A>,
                     r: Parser<Character, A -> A>) -> Parser<Character, A> {

  let apply: A -> (A -> A) -> A = { x in { f in f(x) } }
  return apply </> l <*> (r <|> pure { $0 })
}
```

Finally, we can define our operator function. It works by parsing the `next` calculator, optionally followed by the operator and another `next` call. Note that instead of applying evaluate, we have to flip it first (which swaps the order of the parameters). For some operators this isn't necessary (a + b is the same as b + a), but for others it's essential (a - b is not the same as b - a unless b is zero).

```
func op(character: Character,
        evaluate: (Int, Int) -> Int,
        next: Calculator) -> Calculator {

    let withOperator = curry(flip(evaluate)) </ symbol(character) <*> next
    return optionallyFollowed(next, withOperator)
}
```

We now finally have all the ingredients to once again define our complete parser. Note that instead of giving just `pExpression()` to our `testParser` function, we combine it with `eof()`. This makes sure that the parser consumes all the input (an expression followed by the end of the file).

```
func pExpression() -> Calculator {
  return operatorTable.reduce(number, { next, inOp in
     op(inOp.0, inOp.1, next)
  })
}
testParser(pExpression() <* eof(), "10-3*2")

> Success, found 4, remainder: []
```

This parser is much more efficient because it doesn't have to keep parsing the same things over and over again. In the next chapters, we'll use this parsing library to build a small spreadsheet application.