objc ↑↓

Functional Programming in Swift

Chris Eidhof, Florian Kugler, and Wouter Swierstra

Contents

Chapter 1

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? And why bother learning about this in the first place?

It is hard to give a precise definition of functional programming — in the same way, it is hard to give a precise definition of object-oriented programming, or any other programming paradigm for that matter. Instead, we will try to focus on some of the *qualities* that we believe well-designed functional programs in Swift should exhibit:

- **Modularity:** 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; all these pieces can be assembled using function application to define a complete program. Of course, this decomposition of a large program into smaller pieces only works if we can avoid sharing state between the individual components. This brings us to our next point.

- **A Careful Treatment of Mutable State:** Functional programming is sometimes (half-jokingly) referred to as 'value-oriented programming.' Object-oriented programming focuses on the design of classes and objects, each with their own encapsulated state. Functional programming, on the other hand, emphasizes the importance of programming with values, free of mutable state or other side effects. By avoiding mutable state, functional programs can be more easily combined than their imperative or object-oriented counterparts.

- **Types:** Finally, a well-designed functional program makes careful use of *types*. More than anything else, a careful choice of the types of your data and functions will help structure your code. Swift has a powerful type system that, when used effectively, can make your code both safer and more robust.

We feel these are the key insights that Swift programmers may learn from the functional programming community. Throughout this book, we will illustrate each of these points with many examples and case studies.

In our experience, learning to think functionally is not easy. 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! Functional programming is not the *only* way to program in Swift. Instead, we believe that learning about functional programming adds an important new tool to your toolbox that will make you a better developer in any language.

Current Status of the Book

We approached the process of writing this book very much like we would with developing software: you're looking at version 1.0. We wanted to get a first version out around the time Swift hits 1.0, but we will continue to make adjustments, fix bugs, and add new content as the language evolves.

Should you encounter any mistakes or would like to send any other kind of feedback our way, please file an issue in this GitHub repository.

Acknowledgements

We'd like to thank the numerous people who helped shape this book. We wanted to explicitly mention some of them:

Natalye Childress is our copy-editor. She has provided invaluable feedback, not only making sure the language is correct and consistent, but also making sure things are understandable.

Sarah Lincoln has designed the cover and given us feedback on the design and layout of the book.

Wouter would like to thank *Utrecht University* for letting him take time to work on this book.

We would like to thank the beta readers for their feedback during the writing of this book (listed in alphabetical order):

Bang Jun-young, Adrian Kosmaczewski, Alexander Altman, Andrew Halls, Daniel Eggert, Daniel Steinberg, David Hart, David Owens II, Eugene Dorfman, f-dz-v, Henry Stamerjohann, J Bucaran, Jamie Forrest, Jaromir Siska, Jason Larsen, Jesse Armand, John Gallagher, Kaan Dedeoglu, Kare Morstol, Kiel Gillard, Kristopher Johnson, Matteo Piombo, Nicholas Outram, Ole Begemann, Rob Napier, Ronald Mannak, Sam Isaacson, Ssu Jen Lu, Stephen Horne, TJ, Terry Lewis, Tim Brooks, Vadim Shpakovski.

Chris, Florian, and Wouter

Chapter 2

Thinking Functionally

Functions in Swift are *first-class values*, i.e. functions may be passed as arguments to other functions, and functions may return new 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 explain why first-class functions are useful and provide our first example of functional programming in action.

Example: Battleship

We'll introduce first-class functions using a small example: a non-trivial function that you might need to implement if you were writing a Battleship-like game. 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 to us.

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 in Figure 2.1:

The first function we write, `inRange1`, checks that a point is in the grey

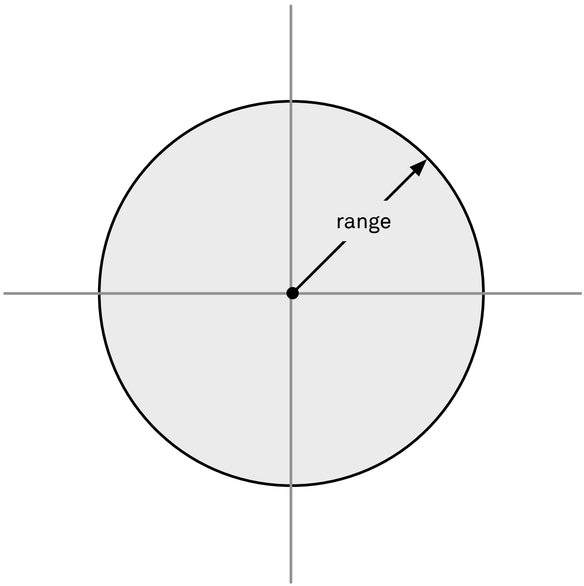

Figure 2.1: The points in range of a ship located at the origin

area in Figure 2.1. 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
}
```

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

Now this works fine, if you assume that we are always located at the origin. But suppose the ship may be at a location, `ownposition`, other than the origin. We can update our visualization in Figure 2.2:

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
    let targetDistance = sqrt(dx * dx + dy * dy)
    return targetDistance <= range
}
```

But now you realize that you also want to avoid targeting ships if they are too close to you. We can update our visualization to illustrate the new situation in Figure 2.3, where we want to target only those enemies that are at least `minimumDistance` away from our current position:

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

```
let minimumDistance: Distance = 2.0
```

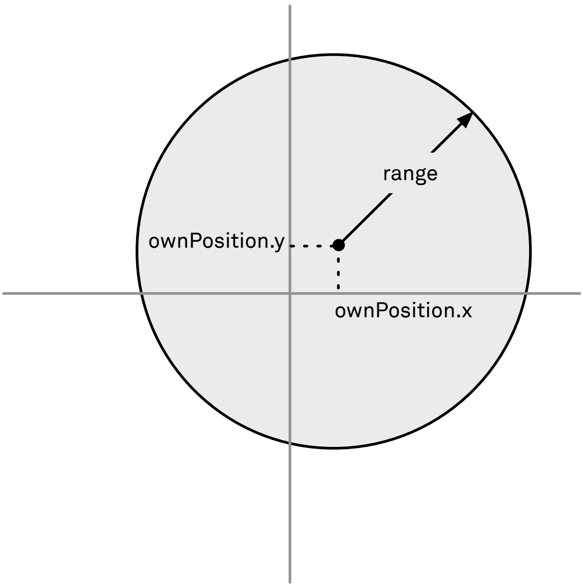

Figure 2.2: Allowing the ship to have its `ownposition`

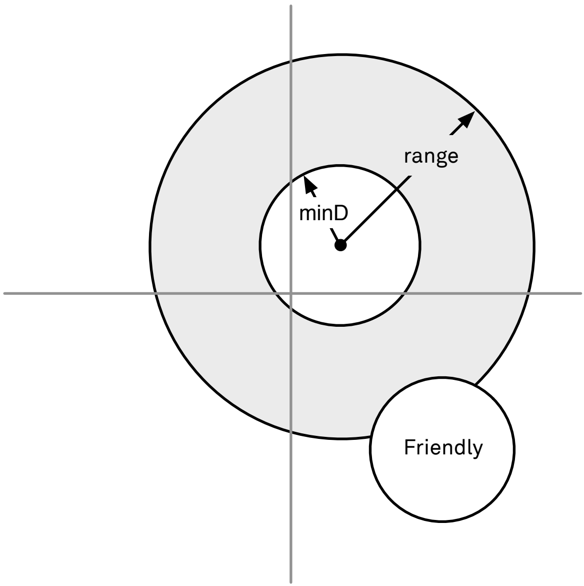

Figure 2.3: Avoiding engaging enemies too close to the ship

```
func inRange3(target: Position, ownPosition: Position,
              range: Distance) -> Bool {

    let dx = ownPosition.x - target.x
    let dy = ownPosition.y - target.y
    let targetDistance = sqrt(dx * dx + dy * dy)
    return targetDistance <= range
           && targetDistance >= minimumDistance
}
```

Finally, you also need to avoid targeting ships that are too close to one of your other ships. Once again, we can visualize this in Figure 2.4:

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 targetDistance = sqrt(dx * dx + dy * dy)
    let friendlyDx = friendly.x - target.x
    let friendlyDy = friendly.y - target.y
    let friendlyDistance = sqrt(friendlyDx * friendlyDx + friendlyDy * friendly
    return targetDistance <= range
           && targetDistance >= minimumDistance
           && (friendlyDistance >= 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.

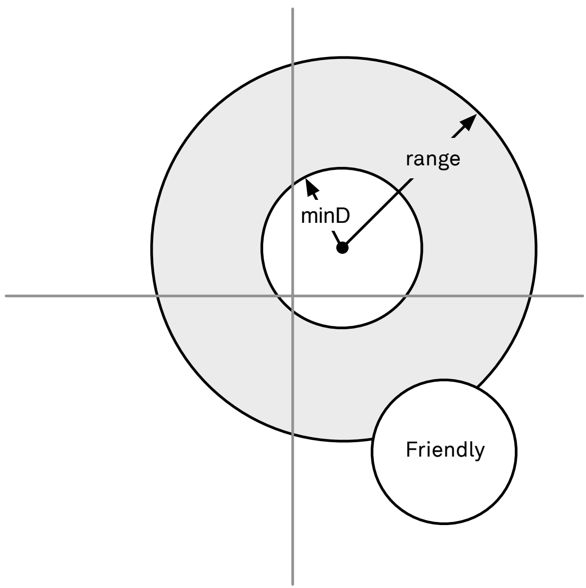

Figure 2.4: avoiding engaging targets too close to friendly ships

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' (or some definition of close and far away). 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 consciously chose 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, integers, or booleans — 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 { p in sqrt(p.x * p.x + p.y * p.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`, which is a function from a point to a boolean value. 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 argument 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 `difference` 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 rangeRegion = difference(circle(range), circle(minimumDistance))
    let targetRegion = shift(ownPosition, rangeRegion)
    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 the 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 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 allow you to 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.

Notes

The code presented here is inspired by the Haskell solution to a problem posed by the United States Advanced Research Projects Agency (ARPA) by

Hudak and Jones (1994).

Objective-C added support for first-class functions when they introduced blocks: you can use functions and closures as parameters, and easily define them inline. However, working with them is not nearly as convenient in Objective-C as it is in Swift, even though they're semantically equivalent.

Historically, the idea of first-class functions can be traced as far back as Church's lambda calculus (Church 1941; Barendregt 1984). Since then, the concept has made its way into numerous (functional) programming languages, including Haskell, OCaml, Standard ML, Scala, and F#.

Chapter 3

Wrapping Core Image

The previous chapter introduced the concept of *higher-order function* and showed how functions can be passed as arguments to other functions. However, the example used there 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 runtime errors. The new API we develop will be safe and modular, exploiting *types* to guarantee the absence of such runtime errors.

Don't worry if you're unfamiliar with Core Image or cannot understand all the details of the code fragments in this chapter. The goal isn't to 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. If you are unfamiliar with Objective-C and programming with dictionaries, you may want to skip this chapter on your first read-through and return to it later.

The Filter Type

One of the key classes in Core Image is the `CIFilter` class, which is 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 key-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 its 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. Later on, this will help us 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 value for the `kCIOutputImageKey` key 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 its parameter:

```
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`). Because of this, 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 are 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 parameters: Parameters = [kCIInputColorKey: color]
        let filter = CIFilter(name:"CIConstantColorGenerator",
                              parameters: parameters)
        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 emphasize 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: overlay
        ]
        let filter = CIFilter(name: "CISourceOverCompositing",
                              parameters: parameters)
        let cropRect = image.extent()
        return filter.outputImage.imageByCroppingToRect(cropRect)
    }
}
```

Here we crop the output image to the size of the input image. This is not strictly necessary, and it depends on how we want the filter to behave. However, 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, `compositeSourceOver(overlay)`, being constructed and subsequently 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: "http://tinyurl.com/m74sldb");
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))
```

However, 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 filter2(filter1(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 `filter1` and `filter2`, 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 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. However, filter composition is a recurring task in an image processing library, so it makes a lot of sense:

```
infix operator >>> { associativity left }

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

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

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

Since we have defined the `>>>` operator as being left-associative we can read the filters that are applied to an image from left to right — like Unix pipes.

The filter composition operation that we have defined is an example of *function composition*. In mathematics, the composition of the two functions `f` and `g`, sometimes written `f ∘ g`, defines a new function mapping an input to `x` to `f(g(x))`. With the exception of the order, 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 case, we first pass the first argument, `1`, which returns a function, which we then apply to the second argument, `2`. Both versions are equivalent: we can define `add1` in terms of `add2`, and vice versa.

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

```
func add2(x: Int) -> Int -> Int {
    return { y in 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 type alias for functional types (as we did for the `Region` and `Filter` types), or write explicit parentheses.

The `add1` and `add2` examples show how 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`.

There is a third way to curry functions in Swift. Instead of constructing the closure explicitly, as we did in the definition of `add2`, we can also define a curried version of `add1` as follows:

```
func add3(x: Int)(y: Int) -> Int {
    return x + y
}
```

Here we have listed the arguments that `add3` expects, one after the other, each surrounded by its own parentheses. To call `add3` we must, however, provide an explicit name for the second argument:

```
add3(1)(y: 2)
```

So why is currying interesting? As we have seen in this book thus far, there are scenarios where you want to pass functions as arguments to other functions. If we have *uncurried* functions, like `add1`, we can only apply a function to *both* its arguments. On the other hand, for a *curried* function, like `add2`, we have a choice: we can apply it to one *or* two arguments. The functions for creating filters that we have defined in this chapter have all been curried — they all expected an additional image argument. By writing our filters in this style, we were able to compose them easily using the

`>>>` operator. Had we instead worked with *uncurried* versions of the same functions, it still would have been possible to write the same filters and filter composition operator, but the resulting code would have been much clunkier.

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 out how higher-order functions and function composition can be used in a more practical case study.

Why go through all this effort? It's true that the Core Image API is already mature and provides all the functionality you might need. But in spite of this, 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. Additionally, composed filters have the exact same type as their building blocks, so you can use them interchangeably.
- **Clarity** — even if you have never used Core Image, you should be able to assemble simple filters using the functions we have defined. To access the results, you don't need to know about special dictionary keys, such as `kCIOutputImageKey`, or worry about initializing 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 4

Map, Filter, Reduce

Functions that take functions as arguments are sometimes called *higher-order* functions. In this chapter, we will tour some of the higher-order functions on arrays from the Swift standard library. By doing so, we will introduce Swift's *generics* and show how to 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 of 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 the individual elements of the array — 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 here — we provide the final (closure) argument to `computeIntArray` after the parentheses containing the other arguments.

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 this:

```
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`, that is, a function that returns an 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 takes a function argument of type `Int -> Bool`. That 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: we can 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 function would be the same again. In fact, the same code will work for *any* type. What we really want to do is write a single generic function that will work for every possible type:

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

The most interesting thing about this piece of code is its type signature. To understand this type signature, it may help you to think of `genericComputeArray<U>` as a family of functions. Each choice of the *type variable* `U` determines a new function. This function takes an array of integers and a function of type `Int -> U` as arguments, and returns an array of type `[U]`.

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

```
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 `T`s and function `f: T -> U`, it will produce a new array of `U`s. This `map` function is even more generic than the `genericComputeArray` function we saw earlier. 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, only it has a more specific type.

There is already a `map` method defined in the Swift standard library in the array type. Instead of writing `map(xs, f)`, we can call `Array`'s `map` function 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 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 the upcoming sections, 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, instead of hardcoding the `.swift` extension, we could pass an additional `String` argument to check against. 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`:

```
func getSwiftFiles2(files: [String]) -> [String] {
    return filter(files) { file in file.hasSuffix(".swift") }
}
```

Just like `map`, the array type already has a `filter` function defined in Swift's standard library. We can call Swift's built-in `filter` function on our `exampleFiles` array, as follows:

```
exampleFiles.filter { file in file.hasSuffix(".swift") }
```

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

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, we will consider a few simple functions before defining a generic function that captures a more 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
}
```

We can use this `sum` function to compute the sum of all the integers in an array:

```
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: String = ""
    for x in xs {
        result += x
    }
    return result
}
```

Or, we can choose to 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: String = "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, and 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`, which 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 thus far using `reduce`. Here are a few examples:

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

Instead of writing a closure, we could have also written just the operator as the last argument. This makes the code even shorter:

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

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

Once again, `reduce` is defined in Swift's standard library as an extension to arrays. From now on, instead of writing `reduce(xs, initialValue, combine)`, we will use `xs.reduce(initialValue, combine)`.

We can use `reduce` to define new generic functions. For example, suppose that we have an array of arrays that we want to flatten into a single array. We could write a function that uses a for loop:

```
func flatten<T>(xss: [[T]]) -> [T] {
    var result : [T] = []
    for xs in xss {
        result += xs
    }
    return result
}
```

Using `reduce`, however, we can write this function as follows:

```
func flattenUsingReduce<T>(xss: [[T]]) -> [T] {
    return reduce(xss, []) { result, xs in result + xs }
}
```

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
        return 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 populations. 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
          return result + "\n" + "\(c.name) : \(c.population)"
      }
```

```
> City: Population
> Paris : 2243000
> Madrid : 3216000
> Berlin : 3397000
```

We start by filtering out those cities that have less than one million inhabitants. We then map our `scale` function over the remaining cities. 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` type 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.

Generics vs. the `Any` Type

Aside from generics, Swift also supports an `Any` type that can represent values of any type. On the surface, this may seem similar to generics. Both the `Any` type and generics can be used to define functions accepting different types of arguments. However, it is very important to understand the difference: generics can be used to define flexible functions, the types of which are still checked by the compiler; the `Any` type can be used to dodge Swift's type system (and should be avoided whenever possible).

Let's consider the simplest possible example, which is a function that does nothing but return its argument. Using generics, we might write the following:

```
func noOp<T>(x: T) -> T {
    return x
}
```

Using the `Any` type, we might write the following:

```
func noOpAny(x: Any) -> Any {
    return x
}
```

Both `noOp` and `noOpAny` will accept any (non-functional) argument. The crucial difference is what we know about the value being returned. In the definition of `noOp`, we can clearly see that the return value is the same as the input value. This is not the case for `noOpAny`, which may return a value of any type — even a type different from the original input. We might also give the following, erroneous definition of `noOpAny`:

```
func noOpAnyWrong(x: Any) -> Any {
    return 0
}
```

Using the `Any` type evades Swift's type system. However, trying to return `0` in the body of the `noOp` function defined using generics will cause a type error. Furthermore, any function that calls `noOpAny` does not know to which type the result must be cast. There are all kinds of possible runtime exceptions that may be raised as a result.

Finally, the *type* of a generic function is extremely informative. Consider the following generic version of the function composition operator,

`>>>`, that we defined in the chapter Wrapping Core Image:

```
infix operator >>> { associativity left }
func >>> <A, B, C>(f: A -> B, g: B -> C) -> A -> C {
    return { x in g(f(x)) }
}
```

The type of this function is so generic that it completely determines how the *function itself* is defined. We'll try to give an informal argument here. We need to produce a value of type `C`. The only way to get our hands on a value of type `C` is by applying the function `g` to a value of type `B`. As there is nothing else we know about `C`, there is no other possible value that we can return. Similarly, the only way to produce a `B` is by applying `f` to a value of type `A`. The only value of type `A` that we have is the final argument to our operator. Hence, this definition of function composition is the only possible function that has this generic type.

In the same way, we can define a generic function that curries any function expecting a tuple of two arguments, thereby producing the corresponding curried version:

```
func curry<A, B, C>(f: (A, B) -> C) -> A -> B -> C {
    return { x in { y in f(x, y) } }
}
```

We no longer need to define two different versions of the same function, the curried and the uncurried, as we did in the last chapter. Instead, generic functions such as `curry` can be used to transform *functions* — computing the curried version from the uncurried. Once again, the type of this function is so generic that it (almost) gives a complete specification: there really is only one sensible implementation.

Using generics allows you to write flexible functions without compromising type safety; if you use the `Any` type, you're pretty much on your own.

Notes

The history of generics traces back to Strachey (2000), Girard's *System F* (1972), and Reynolds (1974). Note that these authors refer to generics as (parametric) polymorphism, a term that is still used in many other functional languages. Many object-oriented languages use the term polymorphism to refer to implicit casts arising from subtyping, so the term generics was introduced to disambiguate between the two concepts.

The process that we sketched informally above, motivating why there can only be one possible function with the generic type `(f: A -> B, g: B -> C) -> A -> C`, can be made mathematically precise. This was first done by Reynolds (1983); later Wadler (1989) referred to this as *Theorems for free!* — emphasizing how you can compute a theorem about a generic function from its type.

Chapter 5

Optionals

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 within the functional programming paradigm.

Case Study: Dictionaries

In addition to arrays, Swift has special support for working with *dictionaries*. A dictionary is a collection of key-value pairs, and it provides 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]
```

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 2,243,000 inhabitants.

As with arrays, the `Dictionary` type is generic. The type of dictionaries takes two type parameters, corresponding to the types of the stored keys and stored values, respectively. In our example, the city dictionary has type `Dictionary<String, Int>`. There is also a shorthand notation, `[String: Int]`.

We can look up 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 the optional type `Int?`. A value of type `Int?` is either an `Int` or a special 'missing' value, `nil`.

We can check whether or not the lookup was successful:

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

If `madridPopulation` is not `nil`, then the 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 multiply it 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 successful, we can use the variable `madridPopulation: Int` 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, thereby avoiding runtime errors. Unchecked usage of the forced unwrapping of optional types or Swift's implicitly unwrapped optionals can be a bad code smell, indicating the possibility of runtime errors.

Swift also provides a safer alternative to the `!` operator, which requires an additional default value to return when applied to `nil`. Roughly speaking, it can be defined as follows:

```
infix operator ??

func ??<T>(optional: T?, defaultValue: T) -> T {
    if let x = optional {
        return x
    } else {
        return defaultValue
    }
}
```

The `??` operator checks whether or not its optional argument is `nil`. If it is, it returns its `defaultValue` argument; otherwise, it returns the optional's underlying value.

There is one problem with this definition: the `defaultValue` may be evaluated, regardless of whether or not the optional is `nil`. This is usually undesirable behavior: an if-then-else statement should only execute *one* of its branches, depending on whether or not the associated condition is true. Similarly, the `??` operator should only evaluate the `defaultValue` argument when the optional argument is `nil`. We can resolve this as follows:

```
func ??<T>(optional: T?, defaultValue: () -> T) -> T {
    if let x = optional {
        return x
    } else {
        return defaultValue()
    }
}
```

Instead of providing a default value of type `T`, we now provide one of type `() -> T`. The code in the `defaultValue` closure is now only executed when we pass it its (void) argument. In this definition, this code is only executed in the else branch, as we intended. The only drawback is that when calling the `??` operator, we need to create an explicit closure for the default value. For example, we would need to write:

```
myOptional ?? { myDefaultValue }
```

The definition in the Swift standard library avoids the need for creating explicit closures by using Swift's `autoclosure` type attribute. This implicitly wraps any arguments to the `??` operator in the required closure. As a result, we can provide the same interface that we initially had, but without requiring the user to create an explicit closure wrapping the `defaultValue` argument. The actual definition used in Swift's standard library is as follows:

```
infix operator ?? { associativity right precedence 110 }

func ??<T>(optional: T?,
           defaultValue: @autoclosure () -> T) -> T {

    if let x = optional {
```

```
        return x
    } else {
        return defaultValue()
    }
}
```

The `??` provides a safer alternative to the forced optional unwrapping, without being as verbose as the optional binding.

Combining Optional Values

Swift's optional values make the possibility of failure explicit. This can be cumbersome, especially when combining multiple 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 runtime 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 More

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:

```swift
func incrementOptional(optional: Int?) -> Int? {
    if let x = optional {
        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, 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:

```swift
func map<T, U>(optional: T?, f: T -> U) -> U? {
    if let x = optional {
        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 not `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:

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

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

Why is this function called map? What does it have to do with array computations? There is a good reason for calling both of these functions map, but we will defer this discussion for the moment. In Chapter 14, we will explain the relation in greater detail.

Optional Binding Revisited

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(optionalX: Int?, optionalY: Int?) -> Int? {
    if let x = optionalX {
        if let y = optionalY {
            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 countries with their capital cities:

```
let capitals = ["France": "Paris", "Spain": "Madrid",
                "The Netherlands": "Amsterdam",
                "Belgium": "Brussels"]
```

In order to write a function that returns the number of inhabitants for the capital of a given country, we use the `capitals` dictionary in conjunction with the `cities` dictionary defined previously. For each dictionary lookup, we have to make sure that it actually returned a result:

```
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 can define a custom operator that captures this pattern:

```
infix operator >>= {}

func >>=<U, T>(optional: T?, f: T -> U?) -> U? {
    if let x = optional {
        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(optionalX: Int?, optionalY: Int?) -> Int? {
    return optionalX >>= { x in
```

```
        optionalY >>= { y in
            x + y
        }
    }
}

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

We do not want to advocate that `>>=` is the 'right' way to combine optional values. Instead, we hope to show that optional binding is not magically built-in to the Swift compiler, but rather a control structure you can implement yourself using a higher-order function.

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 translating 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 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, and an explicit optional type helps protect you from unexpected crashes arising from missing values.

The default zero-like behavior employed by Objective-C has its drawbacks. You may want to distinguish between failure (a key is not in the dictionary) and success-returning nil (a key is in the dictionary, but associated with nil). To do that in Objective-C, you have to use NSNull.

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];
    NSDictionary *attributes = @{ /* ... */ };
    return [[NSAttributedString alloc] initWithString:capital
                                           attributes:attributes];
}
```

While crashes like that don't happen too often, almost every developer has 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 an NSParameterAssert to make sure we crash quickly when the country is nil:

```
- (NSAttributedString *)attributedCapital:(NSString *)country
```

```
{
    NSParameterAssert(country);
    NSString *capital = self.capitals[country];
    NSDictionary *attributes = @{ /* ... */ };
    return [[NSAttributedString alloc] initWithString:capital
                                              attributes:attributes];
}
```

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-empty `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 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 a bit better. Function signatures using optionals explicitly state which values may be `nil`. This is invaluable information when working with other peoples' 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

QuickCheck

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. Additionally, a lot of third-party frameworks (such as Specta, Kiwi, and FBSnapshotTestCase) are already available, and a number of new frameworks are currently being developed in Swift.

All of these frameworks follow a similar pattern: Tests typically consist of some fragment of code, together with an expected result. The code is then executed, and its result is compared to the expected result mentioned in the test. Different libraries test at different levels — some 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 functions.

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 could choose to test that the addition is commutative — in other words, that `a + b` is equal to `b + a`. To test this, we could write a test case that verifies that `42 + 7` is equal to `7 + 42`.

QuickCheck (Claessen and Hughes 2000) is a Haskell library for random testing. Instead of writing individual unit tests, each of which tests that a function is correct for some particular input, QuickCheck allows you to describe abstract *properties* of your functions and *generate* tests to verify these properties. In this chapter, we'll build a (partial) Swift port of QuickCheck.

This is best illustrated with an example. Suppose we want to verify that addition is a commutative operation. To do so, we start by writing a function that checks whether `x + y` is equal to `y + x` for the 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:

```
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 handwritten unit tests.

Of course, not all tests pass. For example, we can define a statement that describes that 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: (0, 1)
> ()
```

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.
> ()
```

Of course, there are many other similar properties of standard arithmetic that we can test. We will cover more interesting tests and properties shortly. Before we do so, however, we will give some more details about how QuickCheck is implemented.

Building QuickCheck

In order to build our Swift implementation of QuickCheck, we will need to do a couple of things.

- First, we need a way to generate random values for different types.
- Using these random value generators, we need to implement the `check` function, which passes random values to its argument property.
- If a test fails, we would like to make the test input as small as possible. 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. This protocol contains only one function, `arbitrary`, which returns a value of type `Self`, i.e. an instance of the class or struct that implements the `Arbitrary` protocol:

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

So 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()
```

```
> 2158783973
```

To generate random strings, we need to do a little bit more work. We start off by generating random characters:

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

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

Then, we generate a random length between 0 and 40 — `x` — using the `random` function defined below. Then, we generate `x` random characters, and reduce them into a string. Note that we currently only generate capital letters as random characters. In a production library, we should generate longer strings that contain arbitrary characters:

```
func tabulate<A>(times: Int, f: Int -> A) -> [A] {
    return Array(0..<times).map(f)
}

func random(#from: Int, #to: Int) -> Int {
    return from + (Int(arc4random()) % (to-from))
}

extension String: Arbitrary {
    static func arbitrary() -> String {
        let randomLength = random(from: 0, to: 40)
        let randomCharacters = tabulate(randomLength) { _ in
            Character.arbitrary()
        }
        return reduce(randomCharacters, "") { $0 + String($1) }
    }
}
```

We use the `tabulate` function to fill an array with the numbers from `0` to `times-1`. By using the `map` function, we then generate an array with the values `f(0)`, `f(1)`, ..., `f(times-1)`. The `arbitrary` extension to `String` uses the `tabulate` function to populate an array of random characters.

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()
```

```
> XMVDXQEIRYNRJTWELHESXHIGPSPOFETEEX
```

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.")
}
```

We could have chosen to use a more functional style by writing this function using `reduce` or `map`, rather than a `for` loop. In this example, however, `for` loops make perfect sense: we want to iterate an operation a fixed number of times, stopping execution once a counterexample has been found — and `for` loops are perfect for that.

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

```
func area(size: CGSize) -> CGFloat {
    return size.width * size.height
}

check1("Area should be at least 0") { size in area(size) >= 0 }
```

```
> "Area should be at least 0" doesn't hold: (-459.570969794777,4403.85297392585)
> ()
```

Here we can see a good example of when QuickCheck can be very useful: it finds an edge case for us. If a size has exactly one negative component, our `area` function will return a negative number. When used as part of a `CGRect`, a `CGSize` can have negative values. When writing ordinary unit tests, it is easy to oversee this case, because sizes usually only have positive components.

Making Values Smaller

If we run our `check1` function on strings, we might receive a rather long failure message:

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

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

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 condition on arrays or dictionaries that fails for some unclear reason — diagnosing why a test is failing is much easier with a minimal counterexample. In principle, the user could try to trim the input that triggered the failure and attempt 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
                            : dropFirst(self)
    }
}
```

To use the `Smaller` protocol in the `check` function, we 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, which takes a condition and 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({ !prop($0) }, value) {
                $0.smaller()
            }
            println("\"\(message)\" doesn't hold: \(smallerValue)")
            return
        }
    }
    println("\"\(message)\" passed \(numberOfIterations) tests.")
}
```

This function is doing quite a bit: generating random input values, checking whether they satisfy the `property` argument, and repeatedly shrinking a counterexample, once one is found. One advantage of defining the repeated shrinking using `iterateWhile`, rather than a separate while loop, is that the control flow of this piece of code stays reasonably simple.

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. Before we can implement `Arbitrary`, we first have to implement `Smaller`. As a first step, we provide a simple definition that drops the first element in the array:

```
extension Array: Smaller {
    func smaller() -> [T]? {
        if !self.isEmpty {
```

```
            return Array(dropFirst(self))
        }
        return nil
    }
}
```

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 tabulate(randomLength) { _ in return X.arbitrary() }
}
```

Now what we'd like to do is define an extension that uses the `arbitraryArray` 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 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 explicitly pass the required information as an argument. By doing so, we have a bit more flexibility. We no longer rely on Swift to *infer* the required information, but instead have complete control over this ourselves.

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 Arbitrary instance, as is the case with arrays, we can overload the check function and construct the desired ArbitraryI struct ourselves:

```
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 be 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.
> ()
```

Using QuickCheck

Somewhat counterintuitively, there is strong evidence to suggest that testing technology influences the design of your code. People who rely on *test-driven design* use tests not only to verify that their code is correct. Instead, they also report that by writing your code in a test-driven fashion, the design of the code gets simpler. This makes sense — if it is easy to write a test for a class without having a complicated setup procedure, it means that the class is nicely decoupled.

For QuickCheck, the same rules apply. It will often not be easy to take existing code and add QuickCheck tests as an afterthought, particularly when you have an existing object-oriented architecture that relies heavily on other classes or makes use of mutable state. However, if you start by doing test-driven development using QuickCheck, you will see that it strongly influences the design of your code. QuickCheck forces you to think of the abstract properties that your functions must satisfy and allows you to give a high-level specification. A unit test can assert that `3 + 0` is equal to `0 + 3`; a QuickCheck property states more generally that addition is a commutative operation. By thinking about a high-level QuickCheck specification first, your code is more likely to be biased toward modularity and *referential transparency* (which we will cover in the next chapter). QuickCheck does not work as well on stateful functions or APIs. As a result, writing your tests up front with QuickCheck will help keep your code clean.

Next Steps

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

- 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 shrunken 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, and we could use that same technique here.

- The `Arbitrary` instances are quite simple. For different data types, we might want to have more complicated arbitrary instances. For example, when generating arbitrary enum values, we could generate certain cases with different frequencies. We could also generate constrained values, such as sorted or non-empty arrays. When writing multiple `Arbitrary` instances, it's possible to define some helper functions that aid us in writing these instances.
- Classify the generated test data: if we generate a lot of arrays of length one, we could classify this as a 'trivial' test case. The Haskell library has support for classification, so these ideas could be ported directly.
- We might want better control of the size of the random input that is generated. In the Haskell version of QuickCheck, the `Arbitrary` protocol takes an additional size argument, limiting the size of the random input generated; the `check` function than starts testing 'small' values, which correspond to small and fast tests. As more and more tests pass, the `check` function increases the size to try and find larger, more complicated counterexamples.
- We might also want to initialize the random generator with an explicit seed, and make it possible to replay the generation of test cases. This will make it easier to reproduce failing tests.

Obviously, that's not everything; there are many other small and large things that could be improved upon to make this into a full library.

Chapter 7

The Value of Immutability

Swift has several mechanisms for controlling how values may change. In this chapter, we will explain how these different mechanisms work, distinguish between value types and reference types, and argue why it is a good idea to limit the usage of mutable state.

Variables and References

In Swift, there are two ways to initialize a variable, using either `var` or `let`:

```
var x: Int = 1
let y: Int = 2
```

The crucial difference is that we can assign new values to variables declared using `var`, whereas variables created using `let` *cannot* change:

```
x = 3 // This is fine
y = 4 // This is rejected by the compiler
```

We will refer to variables declared using a `let` as *immutable* variables; variables declared using a `var`, on the other hand, are said to be *mutable*.

Why — you might wonder — would you ever declare an immutable variable? Doing so limits the variable's capabilities. A mutable variable is strictly more versatile. There is a clear case for preferring `var` over `let`. Yet in this section, we want to try and argue that the opposite is true.

Imagine having to read through a Swift class that someone else has written. There are a few methods that all refer to an instance variable with some meaningless name, say `x`. Given the choice, would you prefer `x` to be declared with a `var` or a `let`? Clearly declaring `x` to be immutable is preferable: you can read through the code without having to worry about what the *current* value of `x` is, you're free to substitute `x` for its definition, and you cannot invalidate `x` by assigning it some value that might break invariants on which the rest of the class relies.

Immutable variables may not be assigned a new value. As a result, it is *easier* to reason about immutable variables. In his famous paper, "Go To Statement Considered Harmful," Edgar Dijkstra writes:

> *My... remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed.*

Dijkstra goes on to argue that the mental model a programmer needs to develop when reading through structured code (using conditionals, loops, and function calls, but not goto statements) is simpler than spaghetti code full of gotos. We can take this discipline even further and eschew the use of mutable variables: `var` considered harmful.

Value Types vs. Reference Types

The careful treatment of mutability is not present only in variable declarations. Swift distinguishes between *value* types and *reference* types. The canonical examples of value and reference types are structs and classes, respectively. To illustrate the difference between value types and reference types, we will define the following struct:

```
struct PointStruct {
```

```
    var x: Int
    var y: Int
}
```

Now consider the following code fragment:

```
var structPoint = PointStruct(x: 1, y: 2)
var sameStructPoint = structPoint
sameStructPoint.x = 3
```

What are the values of `structpoint` and `sameStructPoint` after executing this code? Clearly, `sameStructPoint` should be equal to `PointStruct(x: 3, y: 2)`. But what about `structPoint`? Does the assignment to `sameStructPoint` modify the original `structPoint`? This is where the distinction between value types and reference types is important: when assigned to a new variable, such as `sameStructPoint`, value types are copied. In this example, the assignment to `sameStructPoint.x` does *not* update the original `strutPoint`, because structs are reference types. We could, instead, declare a *class* for points:

```
class PointClass {
    var x: Int
    var y: Int

    init(x: Int, y: Int) {
        self.x = x
        self.y = y
    }
}
```

Then we can adapt our code fragment to use this class instead:

```
var classPoint = PointClass(x: 1, y: 2)
var sameClassPoint = classPoint
sameClassPoint.x = 3
```

Now the assignment, `sameClassPoint.x`, modifies both `classPoint` and `sameClassPoint`, because classes are *reference* types. The distinction between value types and reference types is extremely important — you need to understand this distinction to predict how assignments modify data and which code may be affected by such changes.

The difference between value types and reference types is also apparent when calling functions. Consider the following (somewhat contrived) function that always returns the origin:

```
func setStructToOrigin(var point: PointStruct) -> PointStruct {
    point.x = 0
    point.y = 0
    return point
}
```

We use this function to compute a point:

```
var structOrigin: PointStruct = setStructToOrigin(structPoint)
```

What is the value of `structPoint` after this function call? Does the call to `setStructToOrigin` modify the original `structPoint` or not? All value types, such as structs, are copied when passed as function arguments. In this example, the original `structPoint` is unmodified after the call to `setStructToOrigin`.

Now suppose we had written the following function, operating on classes rather than structs:

```
func setClassToOrigin(point: PointClass) -> PointClass {
    point.x = 0
    point.y = 0
    return point
}
```

Now the following function call *would* modify the `classPoint`:

```
var classOrigin = setClassToOrigin(classPoint)
```

When assigned to a new variable or passed to a function, value types are *always* copied, whereas reference types are *not*. Instead, a reference to the existing object or instance is used. Any changes to this reference will also mutate the original object or instance.

Andy Matuschak provides some very useful intuition for the difference between value types and reference types in his article for objc.io.

Structs are not the only value type in Swift. In fact, almost all the types in Swift are value types, including arrays, dictionaries, numbers, booleans, tuples, and enums (the latter will be covered in the coming chapter). Classes are the exception, rather than the rule. This is one example of how Swift is moving away from object-oriented programming in favor of other programming paradigms.

We will discuss the relative merits of classes and structs later on in this section; before we do so, we want to briefly discuss the interaction between the different forms of mutability that we have seen thus far.

Structs and Classes: Mutable or Not?

In the examples above, we have declared all our points and their fields to be mutable, using `var` rather than `let`. The interaction between compound types, such as structs and classes, and the `var` and `let` declarations, requires some explanation.

Suppose we create the following immutable `PointStruct`:

```
let immutablePoint = PointStruct(x: 0, y: 0)
```

Of course, assigning a new value to this `immutablePoint` is not accepted:

```
immutablePoint = PointStruct(x: 1, y: 1) // Rejected
```

Similarly, trying to assign a new value to one of the point's properties is also rejected, although the properties in `PointStruct` have been defined as `var`, since `immutablePoint` is defined using `let`:

```
immutablePoint.x = 3 // Rejected
```

However, if we would have declared the point variable as mutable, we could change its components after initialization:

```
var mutablePoint = PointStruct(x: 1, y: 1)
mutablePoint.x = 3;
```

If we declare the `x` and `y` properties within the struct using the `let` keyword, then we can't ever change them after initialization, no matter whether the variable holding the point instance is mutable or immutable:

```
struct ImmutablePointStruct {
    let x: Int
    let y: Int
}

var immutablePoint2 = ImmutablePointStruct(x: 1, y: 1)

immutablePoint2.x = 3 // Rejected!
```

Of course, we can still assign a new value to `immutablePoint2`:

```
immutablePoint2 = ImmutablePointStruct(x: 2, y: 2)
```

Objective-C

The concept of mutability and immutability should already be familiar to many Objective-C programmers. Many of the data structures provided by Apple's Core Foundation and Foundation frameworks exist in immutable and mutable variants, such as `NSArray` and `NSMutableArray`, `NSString` and `NSMutableString`, and others. Using the immutable types is the default choice in most cases, just as Swift favors value types over reference types.

In contrast to Swift, however, there is no foolproof way to enforce immutability in Objective-C. We could declare the object's properties as read-only (or only expose an interface that avoids mutation), but this will not stop us from (unintentionally) mutating values internally after they have been initialized. When working with legacy code, for instance, it is all too easy to break assumptions about mutability that cannot be enforced by the compiler. Without checks by the compiler, it is very hard to enforce any kind of discipline in the use of mutable variables.

Discussion

In this chapter, we have seen how Swift distinguishes between mutable and immutable values, and between value types and reference types. In this final section, we want to explain *why* these are important distinctions.

When studying a piece of software, *coupling* measures the degree to which individual units of code depend on one another. Coupling is one of the single most important factors that determines how well software is structured. In the worst case, all classes and methods refer to one another, sharing numerous mutable variables, or even relying on exact implementation details. Such code can be very hard to maintain or update: instead of understanding or modifying a small code fragment in isolation, you constantly need to consider the system in its totality.

In Objective-C and many other object-oriented languages, it is common for class methods to be coupled through shared instance variables. As a result, however, mutating the variable may change the behavior of the class's methods. Typically, this is a good thing — once you change the data stored in an object, all its methods may refer to its new value. At the same time, however, such shared instance variables introduce coupling between all the class's methods. If any of these methods or some external function invalidate the shared state, all the class's methods may exhibit buggy behavior. It is much harder to test any of these methods in isolation, as they are now coupled to one another.

Now compare this to the functions that we tested in the QuickCheck chapter. Each of these functions computed an output value that *only* depended on the input values. Such functions that compute the same output for equal inputs are sometimes called *referentially transparent*. By definition, referentially transparent methods are loosely coupled from their environments: there are no implicit dependencies on any state or variables, aside from the function's arguments. Consequently, referentially transparent functions are easier to test and understand in isolation. Furthermore, we can compose, call, and assemble functions that are referentially transparent without losing this property. Referential transparency is a guarantee of modularity and reusability.

Referential transparency increases modularity on all levels. Imagine reading through an API, trying to figure out how it works. The documentation may be sparse or out of date. But if you know the API is free of mutable state — all variables are declared using `let` rather than `var` — this is incredibly valuable information. You never need to worry about initializing objects or processing commands in exactly the right order. Instead, you can just look at types of the functions and constants that the API defines, and how these can be assembled to produce the desired value.

Swift's distinction between `var` and `let` enables programmers not only to distinguish between mutable and immutable data, but also to have the compiler enforce this distinction. Favoring `let` over `var` reduces the complexity of the program — you no longer have to worry about what the current value of mutable variables is, but can simply refer to their immutable definitions. Favoring immutability makes it easier to write referentially transparent functions, and ultimately, reduces coupling.

Similarly, Swift's distinction between value types and reference types encourages you to distinguish between mutable objects that may change and immutable data that your program manipulates. Functions are free to copy, change, or share values — any modifications will only ever affect their local copies. Once again, this helps write code that is more loosely coupled, as any dependencies resulting from shared state or objects can be eliminated.

Can we do without mutable variables entirely? Pure programming languages, such as Haskell, encourage programmers to avoid using mutable state altogether. There are certainly large Haskell programs that do not use any mutable state. In Swift, however, dogmatically avoiding `var` at all costs will not necessarily make your code better. There are plenty of situations where a function uses some mutable state internally. Consider the following example function that sums the elements of an array:

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

```
    }
    return result
}
```

The `sum` function uses a mutable variable, `result`, that is repeatedly updated. Yet the *interface* exposed to the user hides this fact. The `sum` function is still referentially transparent, and arguably easier to understand than a convoluted definition avoiding mutable variables at all costs. This example illustrates a *benign* usage of mutable state.

Such benign mutable variables have many applications. Consider the `qsort` method defined in the QuickCheck chapter:

```
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)
}
```

Although this method mostly avoids using mutable references, it does not run in constant memory. It allocates new arrays, `lesser` and `greater`, which are combined to produce the final result. Of course, by using a mutable array, we can define a version of Quicksort that runs in constant memory and is still referentially transparent. Clever usage of mutable variables can sometimes improve performance or memory usage.

In summary, Swift offers several language features specifically designed to control the usage of mutable state in your program. It is almost impossible to avoid mutable state altogether, but mutation is used excessively and unnecessarily in many programs. Learning to avoid mutable state and objects whenever possible can help reduce coupling, thereby improving the structure of your code.

Chapter 8

Enumerations

Throughout this book, we want to emphasize the important role *types* play in the design and implementation of Swift applications. In this chapter, we will describe Swift's *enumerations*, which enable you to craft precise types representing the data your application uses.

Introducing Enumerations

When creating a string, it is important to know its character encoding. In Objective-C, an `NSString` object can have several possible encodings:

```
enum NSStringEncoding {
    NSASCIIStringEncoding = 1,
    NSNEXTSTEPStringEncoding = 2,
    NSJapaneseEUCStringEncoding = 3,
    NSUTF8StringEncoding = 4,
    // ...
}
```

Each of these encodings is represented by a number; the `enum` allows programmers to assign meaningful names to the integer constants associated with particular character encoding.

There are some drawbacks to the enumeration declarations in Objective-C and other C dialects. Most notably, the type *NSStringEncoding* is not precise enough — there are integer values, such as 16, that do not correspond to a valid encoding. Furthermore, because all enumerated types are represented by integers, we can compute with them *as if they are numbers*, which is also a disadvantage:

```
NSAssert(NSASCIIStringEncoding + NSNEXTSTEPStringEncoding
        == NSJapaneseEUCStringEncoding, @"Adds up...");
```

Who would have thought that `NSASCIIStringEncoding + NSNEXTSTEPStringEncoding` is equal to `NSJapaneseEUCStringEncoding`? Such expressions are clearly nonsense, yet they are happily accepted by the Objective-C compiler.

Throughout the examples we have seen so far, we have used Swift's *type system* to catch such errors. Simply identifying enumerated types with integers is at odds with the one of core tenets of functional programming in Swift: using types effectively to rule out invalid programs.

Swift also has an `enum` construct, but it behaves very differently from the one you may be familiar with from Objective-C. We can declare our own enumerated type for string encodings as follows:

```
enum Encoding {
    case ASCII
    case NEXTSTEP
    case JapaneseEUC
    case UTF8
}
```

We have chosen to restrict ourselves to the first four possibilities defined in the `NSStringEncoding` enumeration listed above — there are many common encodings that we have not incorporated in this definition. This Swift enumeration declaration is for the purpose of illustration only. The `Encoding` type is inhabited by four possible values: `ASCII`, `NEXTSTEP`, `JapaneseEUC`, and `UTF8`. We will refer to the possible values of an enumeration as *member values*, or *members* for short. In a great deal of literature, such enumerations are sometimes called *sum types*. Throughout this book, however, we will use Apple's terminology.

In contrast to Objective-C, the following code is *not* accepted by the compiler:

```
let myEncoding = Encoding.ASCII + Encoding.UTF8
```

Unlike Objective-C, enumerations in Swift create new types, distinct from integers or other existing types.

We can define functions that calculate with encodings using `switch` statements. For example, we may want to compute the `NSStringEncoding` corresponding to our encoding enumeration:

```
func toNSStringEncoding(encoding: Encoding) -> NSStringEncoding {
    switch encoding {
        case Encoding.ASCII:
            return NSASCIIStringEncoding
        case Encoding.NEXTSTEP:
            return NSNEXTSTEPStringEncoding
        case Encoding.JapaneseEUC:
            return NSJapaneseEUCStringEncoding
        case Encoding.UTF8:
            return NSUTF8StringEncoding
    }
}
```

This definition defines which value to return for each of our `Encoding` types. Note that we have one branch for each of our four different encoding schemes. If we leave any of these branches out, the Swift compiler warns us that the `toNSStringEncoding` function's switch statement is not complete. Once again, this static check is not present in the enumerations used in Objective-C.

Of course, we can also define a function that works in the opposite direction, creating an `Encoding` from an `NSStringEncoding`:

```
func createEncoding(enc: NSStringEncoding) -> Encoding? {
```

```
    switch enc {
        case NSASCIIStringEncoding:
            return Encoding.ASCII
        case NSNEXTSTEPStringEncoding:
            return Encoding.NEXTSTEP
        case NSJapaneseEUCStringEncoding:
            return Encoding.JapaneseEUC
        case NSUTF8StringEncoding:
            return Encoding.UTF8
        default:
            return nil
    }
}
```

As we have not modeled all possible `NSStringEncoding` values in our little `Encoding` enumeration, the `createEncoding` function returns an optional `Encoding` value. If none of the first four cases succeed, the `default` branch is selected, which returns `nil`.

Of course, we do not need to use switch statements to work with our `Encoding` enumeration. For example, if we want the localized name of an encoding, we can compute it as follows:

```
func localizedEncodingName(encoding: Encoding) -> String {
    return String.localizedNameOfStringEncoding(
            toNSStringEncoding(encoding))
}
```

Associated Values

So far, we have seen how Swift's enumerations can be used to describe a choice between several different alternatives. The `Encoding` enumeration provided a safe, typed representation of different string encoding schemes. There are, however, many more applications of enumerations.

Suppose that we want to write Swift wrappers around the existing Objective-C functions to read and write files. Functions that might return an error, such as the `NSString`'s initializer below, can be a bit clunky. In addition to the path of the file to open, as well as the string encoding, it requires a third argument: a pointer to memory for potential error messages. In Swift, we can use an optional type to provide a slightly simpler interface:

```
func readFile1(path: String, encoding: Encoding) -> String? {
    var maybeError: NSError? = nil
    let maybeString = NSString(contentsOfFile: path,
                          encoding: toNSStringEncoding(encoding),
                          error: &maybeError)
    return maybeString
}
```

The `readFile1` function returns an optional string. It simply calls the initializer, passing the address for an `NSError` object. If the call to the initializer succeeds, the resulting string is returned; upon failure, the whole function returns `nil`.

This interface is a bit more precise than Objective-C's `stringWithContentsOfFile` — from the type alone, it is clear that this function may fail. There is no temptation for developers to pass null pointers for the `error` argument, ignoring the possibility of failure.

There is one drawback to using Swift's optional type: we no longer return the error message when the file cannot be read. This is rather unfortunate — if a call to `readFile1` fails, there is no way to diagnose what went wrong. Does the file not exist? Is it corrupt? Or do you not have the right permissions?

Ideally, we would like our `readFile` function to return *either* a `String` *or* an `NSError`. Using Swift's enumerations, we can do just that. Instead of returning a `String?`, we will redefine our `readFile` function to return a member of the `ReadFileResult` enumeration. We can define this enumeration as follows:

```
enum ReadFileResult {
    case Success(String)
    case Failure(NSError)
}
```

In contrast to the `Encoding` enumeration, the members of the `OpenFileResult` have *associated values*. The `ReadFileResult` has only two possible member values: `Success` and `Failure`. In contrast to the `Encoding` enumeration, both of these member values carry additional information: the `Success` member has a string associated with it, corresponding to the contents of the file; the `Failure` member has an associated `NSError`. To illustrate this, we can declare an example `Success` member as follows:

```
let exampleSuccess: ReadFileResult = ReadFileResult.Success(
        "File contents goes here")
```

Similarly, to create a `ReadFile` result using the `Failure` member, we would need to provide an associated `NSError` value.

Now we can rewrite our `readFile` function to return a `ReadFileResult`:

```
func readFile(path: String, encoding: Encoding) -> ReadFileResult {
    var maybeError: NSError?
    let maybeString: String? = NSString(contentsOfFile: path,
          encoding: toNSStringEncoding(encoding), error: &maybeError)
    if let string = maybeString {
        return ReadFileResult.Success(string)
    } else if let error = maybeError {
        return ReadFileResult.Failure(error)
    } else {
        assert(false, "The impossible occurred")
    }
}
```

Instead of returning an optional `String`, we now return either the file contents or an `NSError`. We first check if the string is non-`nil` and return the value. If the string was `nil`, we check if there was an error opening the file and, if so, return a `Failure`. Finally, if both `maybeError` and `maybeString` are `nil` after the call to the initializer, something is very wrong indeed — we assert `false` and should try to give a meaningful error message.

Upon calling `readFile`, you can use a switch statement to determine whether or not the function succeeded:

```
switch readFile("/Users/wouter/fpinswift/README.md", Encoding.ASCII) {
    case let ReadFileResult.Success(contents):
        println("File succesfully opened..")
    case let ReadFileResult.Failure(error):
        println("Failed to open file. Error code: \(error.code)")
}
```

In contrast to the type interface defined by the Objective-C version of `stringWithContentsOfFile`, it is clear from the types alone what to expect when calling `readFile`. There are corner cases in Objective-C that are not clearly defined: what to do if, after calling `stringWithContentsOfFile`, the error message is non-null, but you also have a valid `String`[1]? Or what happens if the string is `nil`, and there is no error message? The `ReadFileResult` type makes it crystal clear what you can expect: a `String` or an `NSError`. You don't have to read any supporting documentation to understand how to treat the result.

Adding Generics

Now that we have a Swift function for reading files, the obvious next challenge is to define a function for writing files. As a first approximation, we might write the following:

```
func writeFile(contents: String,
               path: String, encoding: Encoding) -> Bool {

    return contents.writeToFile(path,
                                atomically: false,
```

[1] See the documentation.

```
                                        encoding: toNSStringEncoding(encoding),
                                        error: nil)
}
```

The `writeFile` function now returns a boolean, indicating whether or not the operation succeeded. Unfortunately, this definition suffers from the same limitations as our earlier version of `readFile`: when `writeFile` fails, we are not returning the `NSError`.

But we now know how to solve this! Our first approach might be to reuse the `ReadFileResult` enumeration to return the error. When `writeFile` succeeds, however, we do not have a string to associate with the `Success` member value. While we could pass some dummy string, doing so usually indicates bad design: our types should be precise enough to prevent us from having to work with such dummy values.

Alternatively, we can define a new enumeration, `WriteFileResult`, corresponding to the two possible cases:

```
enum WriteFileResult {
    case Success
    case Failure(NSError)
}
```

We can certainly write a new version of the `writeFile` function using this enumeration — but introducing a new enumeration for each possible function seems like overkill. Besides, the `WriteFileResult` and `ReadFileResult` have an awful lot in common. The only difference between the two enumerations is the value associated with `Success`. We would like to define a new enumeration that is *generic* in the result associated with `Success`:

```
enum Result<T> {
    case Success(T)
    case Failure(NSError)
}
```

Unfortunately, generic associated values are not supported by the current Swift compiler. But there is a workaround — defining a dummy wrapper `Box<T>`:

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

enum Result<T> {
    case Success(Box<T>)
    case Failure(NSError)
}
```

The `Box` class does not serve any particular purpose, except to hide the associated generic value `T` in the `Success` member.

Now we can use the same result type for both `readFile` and `writeFile`. Their new type signatures would become:

```
func readFile(path: String, encoding: Encoding) -> Result<String>
func writeFile(contents: String,
               path: String, encoding: Encoding) -> Result<()>
```

The `readFile` function returns either a `String` or an `NSError`; the `writeFile` function returns nothing, represented by the void type `()` or an `NSError`.

Optionals Revisited

Under the hood, Swift's built-in optional type is very similar to the `Result` type that we've defined here. The following snippet is taken directly from the Swift standard library:

```
enum Optional<T> {
    case None
    case Some(T)
    // ...
}
```

The optional type just provides some syntactic sugar, such as the postfix `?` notation and optional unwrapping mechanism, to make it easier to use. There is, however, no reason that you couldn't define it yourself.

In fact, we can even define some of the library functions for manipulating optionals on our own `Result` type. For example, our `Result` type also supports a `map` operation:

```
func map<T, U>(f: T -> U, result: Result<T>) -> Result<U> {
    switch result {
        case let Result.Success(box):
            return Result.Success(Box(f(box.unbox)))
        case let Result.Failure(error):
            return Result.Failure(error)
    }
}
```

Similarly, we can redefine the `??` operator to work on our `Result` type. Note that, instead of taking an `autoclosure` argument, we expect a function that handles the `NSError` to produce the desired value of type `T`:

```
func ??<T>(result: Result<T>, handleError: NSError -> T) -> T {
    switch result {
        case let Result.Success(box):
            return box.unbox
        case let Result.Failure(error):
            return handleError(error)
    }
}
```

The Algebra of Data Types

As we mentioned previously, enumerations are often referred to as sum types. This may be a confusing name, as enumerations seem to have no relation to numbers. Yet if you dig a little deeper, you may find that enumerations and tuples have mathematical structure, very similar to arithmetic. To illustrate this, consider the following enumeration:

```
enum Add<T, U> {
```

```
    case InLeft(Box<T>)
    case InRight(Box<U>)
}
```

For any types `T` and `U`, the enumeration `Add<T, U>` consists of either a (boxed) value of type `T`, or a (boxed) value of type `U`. As its name suggests, the `Add` enumeration adds together the members from the types `T` and `U`: if `T` has three members and `U` has seven, `Add<T, U>` will have ten possible members.

In arithmetic, zero is the unit of addition, i.e. `x + 0` is the same as using just `x` for any number `x`. Can we find an enumeration that behaves like zero? Interestingly, Swift allows us to define the following enumeration:

```
enum Zero { }
```

This enumeration is empty — it doesn't have any members. As we hoped, this enumeration behaves exactly like the zero of arithmetic: for any type `T`, the types `Add<T, Zero>` and `T` are isomorphic.

So much for addition — let us now consider multiplication. If we have an enumeration, `T`, with three members, and another enumeration, `U`, with two members, how can we define a compound type, `Times<T, U>`, with six members? To do this, the `Times<T, U>` type should allow you to choose *both* a member of `T` and a member of `U`. In other words, it should correspond to a pair of two values of type `T` and `U` respectively:

```
struct Times<T, U> {
    let fst: T
    let snd: U
}
```

Just as `Zero` was the unit of addition, the void type, `()`, is the unit of `Times`:

```
typealias One = ()
```

It is easy to check that many familiar laws from arithmetic are still valid when read as isomorphisms between types:

- `Times<One, T>` is isomorphic to `T`

- `Times<Zero, T>`is isomorphic to `Zero`
- `Times<T, U>` is isomorphic to `Times<U, T>`

Types defined using enumerations and tuples are sometimes referred to as *algebraic data types*, because they have this algebraic structure, similar to natural numbers.

This correspondence between numbers and types runs much deeper than we have sketched here. Functions can be shown to correspond to exponentiation. There is even a notion of differentiation that can be defined on types!

This observation may not be of much practical value, but it is important to understand that enumerations, like many of Swift's features, are not new, but rather draw on years of research in mathematics and program language design.

Why Use Enumerations?

Working with optionals may still be preferable over the `Result` type that we have defined here, for a variety of reasons: the built-in syntactic sugar can be convenient; the interface you define will be more familiar to Swift developers, as you only rely on existing types instead of defining your own enumeration; and sometimes the `NSError` is not worth the additional hassle of defining an enumeration.

The point we want to make, however, is not that the `Result` type is the best way to handle all errors in Swift. Instead, we hope to illustrate how you can use enumerations to define your own types, tailored to your specific needs. By making these types precise, you can use Swift's type checking to your advantage and prevent many bugs, before your program has been tested or run.

Chapter 9

Purely Functional Data Structures

In the previous chapter, we saw how to use enumerations to define specific types tailored to the application you are developing. In this chapter, we will define *recursive* enumerations and show how these can be used to define data structures that are both efficient and persistent.

Binary Search Trees

Swift does not have a library for manipulating sets, like Objective-C's `NSSet` library. While we could write a Swift wrapper around `NSSet`, like we did for Core Image and the `String` initializer, we will explore a slightly different approach. Our aim is, once again, not to define a complete library for manipulating sets in Swift, but rather to demonstrate how recursive enumerations can be used to define efficient data structures.

In our little library, we will implement the following four operations:

- `emptySet` — returns an empty set
- `isEmptySet` — checks whether or not a set is empty
- `setInsert` — adds an element to an existing set
- `setContains` — checks whether or not an element is in a set

As a first attempt, we may use arrays to represent sets. These four operations are almost trivial to implement:

```
func emptySet<T>() -> Array<T> {
    return []
}

func isEmptySet<T>(set: [T]) -> Bool {
    return set.isEmpty
}

func setInsert<T>(x: T, set:[T]) -> [T] {
    return [x] + set
}

func setContains<T: Equatable>(x: T, set: [T]) -> Bool {
    return contains(set, x)
}
```

While simple, the drawback of this implementation is that many of the operations perform linearly in the size of the set. For large sets, this may cause performance problems.

There are several possible ways to improve performance. For example, we could ensure the array is sorted and use binary search to locate specific elements. Instead, we will define a *binary search tree* to represent our sets. We can build a tree structure in the traditional C style, maintaining pointers to subtrees at every node. However, we can also define such trees directly as an enumeration in Swift, using the same `Box` trick as in the last chapter:

```
enum Tree<T> {
    case Leaf
    case Node(Box<Tree<T>>, T, Box<Tree<T>>)
}
```

This definition states that every tree is either:

- a `Leaf` without associated values, or
- a `Node` with three associated values, which are the left subtree, the value stored at the node, and the right subtree.

Before defining functions on trees, we can write a few example trees by hand:

```
let leaf: Tree<Int> = Tree.Leaf


let five: Tree<Int> = Tree.Node(Box(leaf), 5, Box(leaf))
```

Note: this currently compiles but hangs when run (Xcode 6.1, beta 2).

The `leaf` tree is empty; the `five` tree stores the value `5` at a node, but both subtrees are empty. We can generalize this construction and write a function that builds a tree with a single value:

```
func single<T>(x: T) -> Tree<T> {
    return Tree.Node(Box(Tree.Leaf), x, Box(Tree.Leaf))
}
```

Just as we saw in the previous chapter, we can write functions that manipulate trees using switch statements. As the `Tree` enumeration itself is recursive, it should come as no surprise that many functions that we write over trees will also be recursive. For example, the following function counts the number of elements stored in a tree:

```
func count<T>(tree: Tree<T>) -> Int {
    switch tree {
        case let Tree.Leaf:
            return 0
        case let Tree.Node(left, x, right):
            return 1 + count(left.unbox) +
                       count(right.unbox)
    }
}
```

In the base case for leaves, we can return `0` immediately. The case for nodes is more interesting: we compute the number of elements stored in both subtrees *recursively*. We then return their sum, and add `1` to account for the value `x` stored at this node.

Similarly, we can write an `elements` function that calculates the array of elements stored in a tree:

```
func elements<T>(tree: Tree<T>) -> [T] {
    switch tree {
        case let Tree.Leaf:
            return []
        case let Tree.Node(left, x, right):
            return elements(left.unbox) + [x] +
                   elements(right.unbox)
    }
}
```

Now let's return to our original goal, which is writing an efficient set library using trees. We have obvious choices for the `isEmptySet` and `emptySet` functions:

```
func emptySet<T>() -> Tree<T> {
    return Tree.Leaf
}

func isEmptySet<T>(tree: Tree<T>) -> Bool {
    switch tree {
        case let Tree.Leaf:
            return true
        case let Tree.Node(_, _, _):
            return false
    }
}
```

Note that in the `Node` case for the `isEmptySet` function, we do not need to refer to the subtrees or the value stored at the node, but can immediately return `false`. Correspondingly, we can put wildcard patterns for the three values associated with a `Node`, indicating they are not used in this case branch.

If we try to write naive versions of `setInsert` and `setContains`, however, it seems that we have not gained much. If we restrict ourselves to *binary search trees*, however, we can perform much better. A (non-empty) tree is said to be a binary search tree if all of the following conditions are met:

- all the values stored in the left subtree are *less* than the value stored at the root
- all the values stored in the right subtree are *greater* than the value stored at the root
- both the left and right subtrees are binary search trees

We can write an (inefficient) check to ascertain if a `Tree` is a binary search tree or not:

```
func isBST<T: Comparable>(tree: Tree<T>) -> Bool {
    switch tree {
        case Tree.Leaf:
            return true
        case let Tree.Node(left, x, right):
            let leftElements = elements(left.unbox)
            let rightElements = elements(right.unbox)
            return all(leftElements) { y in y < x }
                   && all(rightElements) { y in y > x }
                   && isBST(left.unbox)
                   && isBST(right.unbox)
    }
}
```

The `all` function checks if a property holds for all elements in an array. It is defined in the appendix of this book.

The crucial property of binary search trees is that they admit an efficient lookup operation, akin to binary search in an array. As we traverse the tree to determine whether or not an element is in the tree, we can rule out (up to) half of the remaining elements in every step. For example, here is one possible definition of the `setContains` function that determines whether or not an element occurs in the tree:

```
func setContains<T: Comparable>(x: T, tree: Tree<T>) -> Bool {
    switch tree {
        case Tree.Leaf:
            return false
        case let Tree.Node(left, y, right) where x == y:
            return true
        case let Tree.Node(left, y, right) where x < y:
            return setContains(x, left.unbox)
        case let Tree.Node(left, y, right) where x > y:
            return setContains(x, right.unbox)
        default:
            assert(false, "The impossible occurred")
    }
}
```

The `setContains` function now distinguishes four possible cases:

- If the `tree` is empty, the `x` is not in the tree and we return false.
- If the `tree` is non-empty and the value stored at its root is equal to `x`, we return true.
- If the `tree` is non-empty and the value stored at its root is greater than `x`, we know that if `x` is in the tree, it must be in the left subtree. Hence, we recursively search for `x` in the left subtree.
- Similarly, if `x` is greater than the value stored at the root, we proceed by searching the right subtree.

Unfortunately, the Swift compiler is not clever enough to see that these four cases cover all the possibilities, so we need to insert a dummy default case.

Insertion searches through the binary search tree in exactly the same fashion:

```swift
func setInsert<T: Comparable>(x: T, tree: Tree<T>) -> Tree<T> {
    switch tree {
        case Tree.Leaf:
            return single(x)
        case let Tree.Node(left, y, right) where x == y:
            return tree
        case let Tree.Node(left, y, right) where x < y:
            return Tree.Node(Box(setInsert(x, left.unbox)),
                             y, right)
        case let Tree.Node(left, y, right) where x > y:
            return Tree.Node(left, y,
                             Box(setInsert(x, right.unbox)))
        default:
            assert(false, "The impossible occurred")
    }
}
```

Instead of checking whether or not the element occurs, `setInsert` finds a suitable location to add the new element. If the tree is empty, it builds a tree with a single element. If the element is already present, it returns the original tree. Otherwise, the `setInsert` function continues recursively, navigating to a suitable location to insert the new element.

The worst-case performance of `setInsert` and `setContains` on binary search trees is still linear — after all, we could have a very unbalanced tree, where every left subtree is empty. More clever implementations, such as 2-3 trees, AVL trees, or red-black trees, avoid this by maintaining the invariant that each tree is suitably balanced. Furthermore, we haven't written a `delete` operation, which would also require rebalancing. These are tricky operations for which there are plenty of well-documented implementations in the literature — once again, this example serves as an illustration of working with recursive enumerations and does not pretend to be a complete library.

Autocompletion Using Tries

Now that we've seen binary trees, this last section will cover a more advanced and purely functional data structure. Suppose that we want to write our own autocompletion algorithm — given a history of searches and the prefix of the current search, we should compute an array of possible completions.

Using arrays, the solution is entirely straightforward:

```
func autocomplete(history: [String],
                  textEntered: String) -> [String] {

    return history.filter { string in
        string.hasPrefix(textEntered)
    }
}
```

Unfortunately, this function is not very efficient. For large histories and long prefixes, it may be too slow. Once again, we could improve performance by keeping the history sorted and using some kind of binary search on the history array. Instead, we will explore a different solution, using a custom data structure tailored for this kind of query.

Tries, also known as digital search trees, are a particular kind of ordered tree. Typically, tries are used to look up a string, which consists of a list of characters. Instead of storing strings in a binary search tree, it can be more efficient to store them in a structure that repeatedly branches over the strings' constituent characters.

Previously, the binary `Tree` type had two subtrees at every node. Tries, on the other hand, do not have a fixed number of subtrees at every node, but instead (potentially) have subtrees for every character. For example, we could visualize a trie storing the string "cat," "car," "cart," and "dog" as follows:

To determine if the string "care" is in the trie, we follow the path from the root, along the edges labeled 'c,' 'a,' and 'r.' As the node labeled 'r' does not have a child labeled with 'e,' the string "care" is not in this trie.

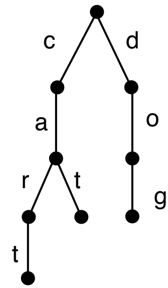

Figure 9.1:

The string “cat” is in the trie, as we can follow a path from the root along edges labeled ‘c,’ ‘a,’ and ‘t.’

How can we represent such tries in Swift? As a first attempt, we write an enumeration storing a dictionary, mapping characters to subtries at every node:

```
enum Trie {
    case Node([Character: Trie])
}
```

There are two improvements we would like to make to this definition. First of all, we need to add some additional information to the node. From the example trie above, you can see that by adding “cart” to the trie, all the prefixes of “cart” — namely “c,” “ca,” and “car” — also appear in the trie. As we may want to distinguish between prefixes that are or are not in the trie, we will add an additional boolean to every node. This boolean indicates whether or not the current string is in the trie. Finally, we can define a generic trie that is no longer restricted to only storing characters. Doing so yields the following definition of tries:

```
enum Trie<T: Hashable> {
```

```
    case Make(Bool, [T: Trie<T>])
}
```

In the text that follows, we will sometimes refer to the keys of type `[T]` as strings, and values of type `T` as characters. This is not very precise — as `T` can be instantiated with a type different than characters, and a string is not the same as `[Character]` — but we hope it does appeal to the intuition of tries storing a collection of strings.

Before defining our autocomplete function on tries, we will write a few simple definitions to warm up. For example, the empty trie consists of a node with an empty dictionary:

```
func empty<T: Hashable>() -> Trie<T> {
    return Trie.Make(false, [T: Trie<T>]())
}
```

If we had chosen to set the boolean stored in the empty trie to true rather than false, the empty string would be a member of the empty trie — which is probably not the behavior that we want.

Next, we define a function to flatten a trie into an array containing all its elements:

```
func elements<T: Hashable>(trie: Trie<T>) -> [[T]] {
    switch trie {
        case let Trie.Make(isElem, rest):
            var result: [[T]] = isElem ? [[]] : []
            for (key, value) in rest {
                result += elements(value).map {xs in
                    [key] + xs
                }
            }
            return result
    }
}
```

This function is a bit tricky. It starts by checking if the current root is marked as a member of the trie or not. If it is, the trie contains the empty key; if it is not, the `result` variable is initialized to the empty array. Next, it traverses the dictionary, computing the elements of the subtries — this is done by the call `elements(value)`. Finally, the 'character' associated with every subtrie is added to the front of the elements of that subtrie — this is taken care of by the `map` function.

Next, we would like to define lookup and insertion functions. Before we do so, however, we will need a few auxiliary functions. We have represented keys as an array. While our tries are defined as (recursive) enumerations, arrays are not. Yet it can still be useful to traverse an array recursively. To make this a bit easier, we define the following extension on arrays:

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

The `decompose` function checks whether or not an array is empty. If it is empty, it returns `nil`; if the array is not empty, it returns a tuple containing both the first element of the array and the tail or remainder of the array, with the first element removed. We can recursively traverse an array by repeatedly calling `decompose` until it returns `nil` and the array is empty.

Using the `decompose` extension on arrays, we can define a lookup function that, given an array of `T`s, traverses a trie to determine whether or not the corresponding key is stored:

```
func lookup<T: Hashable>(key: [T],
                         trie: Trie<T>) -> Bool {

    switch trie {
        case let Trie.Make(isElem, rest):
            if let (head, tail) = key.decompose {
                if let subtrie = rest[head] {
                    return lookup(tail, subtrie)
```

```
            }
        }
        else { return isElem }
    }
    return false
}
```

Here we can distinguish three cases:

- The key is empty — in that case, we return `isElem`, the boolean indicating whether or not the string described by the current node is in the trie or not.
- The key is non-empty — in that case, we look up the subtrie corresponding to the first element of the key. If this also exists, we make a recursive call, looking up the tail of the key in this subtrie.
- The key is non-empty, but the corresponding subtrie does not exist — in that case, we simply return false, as the key is not included in the trie.

We can adapt `lookup` to return the subtrie, containing all the elements that have some prefix:

```
func withPrefix<T: Hashable>(prefix: [T],
                             trie: Trie<T>) -> Trie<T>? {

    switch trie {
        case let Trie.Make(isElem, rest):
            if let (head, tail) = prefix.decompose {
                if let remainder = rest[head] {
                    return withPrefix(tail, remainder)
                }
            } else {
                return trie
            }
    }
```

```
    return nil
}
```

The only difference with the `lookup` function is that we no longer return the `isElem` boolean, but instead return the whole subtrie, containing all the elements with the argument prefix.

Finally, we can redefine our `autocomplete` function to use the more efficient tries data structure:

```
func autocomplete<T: Hashable>(key: [T],
                               trie: Trie<T>) -> [[T]] {

    if let prefixTrie = withPrefix(key, trie) {
        return elements(prefixTrie)
    } else {
        return []
    }
}
```

To compute all the strings in a trie with a given prefix, we simply call the `withPrefix` function and extract the elements from the resulting trie, if it exists. If there is no subtrie with the given prefix, we simply return the empty array.

To complete the library, you may still want to write insertion and deletion functions yourself.

Discussion

These are but two examples of writing efficient, immutable data structures using enumerations. There are many others in Chris Okasaki's *Purely Functional Data Structures* (1999), which is a standard reference on the subject. Interested readers may also want to read Ralph Hinze and Ross Paterson's work on finger trees (2006), which are general-purpose purely functional data structures with numerous applications. Finally, StackOverflow has a fantastic list of more recent research in this area.

Chapter 10

Diagrams

In this chapter, we'll look at a functional way to describe diagrams, and discuss 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 diagram in Figure 10.1. 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 difficult to maintain. For example, what if we wanted to add an extra circle like in Figure 10.2?

Figure 10.1: A simple diagram

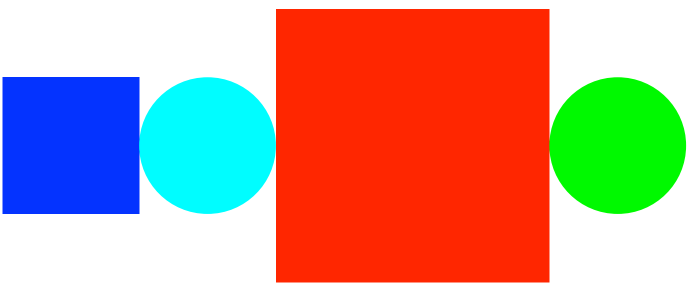

Figure 10.2: Adding an extra circle

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
```

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

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

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`). 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, Figure 10.3 shows 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 in the dictionary, we draw a suitably sized rectangle. We then horizontally concatenate these rectangles with the `hcat` function. Finally, we put the bars and the text below each other using the `---` operator:

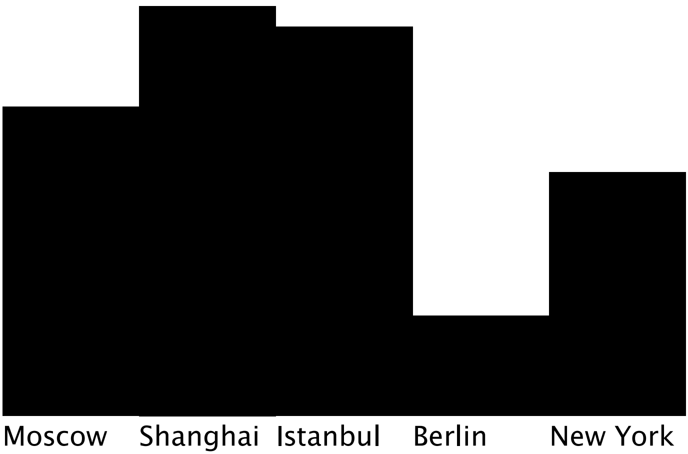

Figure 10.3: A bar graph

```
func barGraph(input: [(String, Double)]) -> Diagram {
    let values: [CGFloat] = input.map { CGFloat($0.1) }
    let nValues = normalize(values)
    let bars = hcat(nValues.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).alignTop()
    })
    return bars --- labels
}
let cities = ["Shanghai": 14.01, "Istanbul": 13.3,
              "Moscow": 10.56, "New York": 8.33, "Berlin": 3.43]
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 data type for these three possibilities:

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

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, as seen in Figure 10.4.

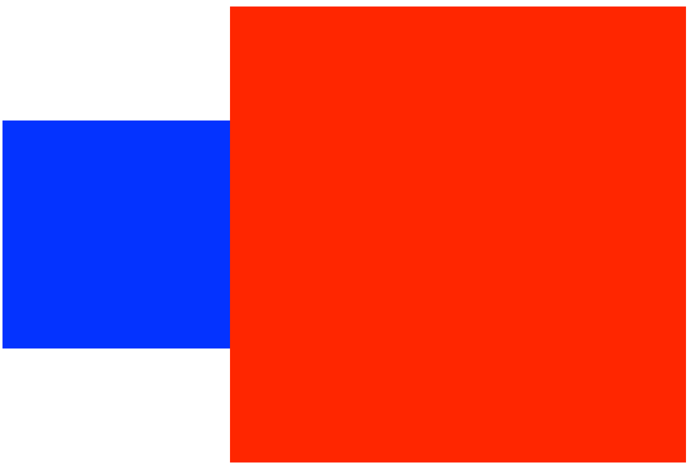

Figure 10.4: Vertical centering

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

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

For example, Figure 10.5 shows a diagram that's top aligned. It is drawn using the following code:

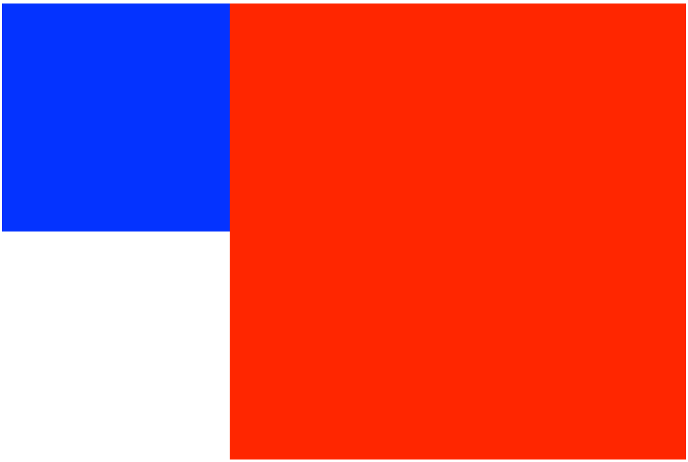

Figure 10.5: Vertical alignment

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

Unfortunately, in the current version of Swift, recursive data types are not allowed. So instead of having a `Diagram` case that contains other `Diagram`s, we have to wrap each recursive `Diagram` with `Box` (defined in Chapter 8)

```
enum Diagram {
    case Prim(CGSize, Primitive)
```

```
    case Beside(Box<Diagram>, Box<Diagram>)
    case Below(Box<Diagram>, Box<Diagram>)
    case Attributed(Attribute, Box<Diagram>)
    case Align(Vector2D, Box<Diagram>)
}
```

The `Attribute` enum is a data type 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, etc.:

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

Calculating and Drawing

Calculating the size for the `Diagram` data type is easy. The only cases that aren't straightforward are 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.unbox.size
        case .Beside(let l, let r):
            let sizeL = l.unbox.size
            let sizeR = r.unbox.size
            return CGSizeMake(sizeL.width + sizeR.width,
                max(sizeL.height, sizeR.height))
        case .Below(let l, let r):
```

```
            let sizeL = l.unbox.size
            let sizeR = r.unbox.size
            return CGSizeMake(max(sizeL.width, sizeR.width),
                sizeL.height+sizeR.height)
        case .Align(_, let r):
            return r.unbox.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)
    return CGRect(origin: rect.origin - space.point, size: size)
}
```

For example, if we fit and center a 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 few 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 this 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 font = NSFont.systemFontOfSize(12)
    let attributes = [NSFontAttributeName: font]
    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.unbox)
    CGContextRestoreGState(context)
```

To draw two diagrams next to each other, we first need to find their respective frames. We create 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.unbox
    let r = right.unbox
    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.unbox
```

```
            let b = bottom.unbox
            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.unbox
            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) {
        if let context = NSGraphicsContext.currentContext() {
            draw(context.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 `NSView`s method, `dataWithPDFInsideRect`, to get the PDF data. This is a nice example of taking existing object-oriented code, and wrapping it in a functional layer:

```
func pdf(diagram: Diagram, width: CGFloat) -> NSData {
    let unitSize = diagram.size
    let height = width * (unitSize.height/unitSize.width)
    let v: Draw = 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 data types and functions, and then build convenience functions on top of them. 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(Box(l), Box(r))
}

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

We can also extend the `Diagram` type and add methods for filling and alignment. We also might have defined these methods as top-level functions instead. 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), Box(self))
    }
```

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

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

Discussion

The code in this chapter is inspired by the Diagrams library for Haskell (Yorgey 2012). Although we can draw simple diagrams, there are many possible improvements and extensions to the library we have presented here. Some 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 (such as rotation), but this is certainly still possible.

When we compare the library that we've built in this chapter to the library in Chapter 2, we can see many similarities. Both take a problem domain (regions and diagrams) and create a small library of functions to describe this domain. Both libraries provide an interface through functions that are highly composable. Both of these little libraries define a *domain-specific language* (or DSL) embedded in Swift. A DSL is a small programming language, tailored to solve a particular problem. You are probably already familiar with lots of DSLs, such as regular expressions, SQL, or HTML — each of these languages is not a general-purpose programming language in which to write *any* application, but instead is more restricted to solve a particular kind of problem. Regular expressions are used for describing patterns or lexers, SQL is used for querying a database, and HTML is used for describing the content of a webpage.

However, there is an important difference between the two DSLs: in the Thinking Functionally chapter, we created functions that return a bool for each position. To draw the diagrams, we built up an intermediate structure, the `Diagram` enum. A *shallow embedding* of a DSL in a general-purpose programming language like Swift does not create any intermediate data structures. A *deep embedding*, on the other hand, explicitly creates an intermediate data structure, like the `Diagram` enumeration described in this chapter. The term 'embedding' refers to how the DSL for regions or diagrams are 'embedded' into Swift. Both have their advantages. A shallow embedding can be easier to write, there is less overhead during execution, and it can be easier to extend with new functions. However, when using a deep embedding, we have the advantage that we can analyze an entire structure, transform it, or assign different meanings to the intermediate data structure.

If we would rewrite the DSL from Chapter 2 to use deep embedding instead, we would need to define an enumeration representing the different functions from the library. There would be members for our primitive regions, like circles or squares, and members for composite regions, such as those formed by intersection or union. We could then analyze and compute with these regions in different ways: generating images, checking whether a region is primitive or not, determining whether or not a given point is in the region, or performing an arbitrary computation over this intermediate data structure. Rewriting the diagrams library to a shallow embedding would be complicated. The intermediate data structure can be inspected, modified, and transformed. To define a shallow embedding, we would need to call Core Graphics directly for every operation that we wish to support in our DSL. It is much more difficult to compose drawing calls than it is to first create an intermediate structure and only render it once the diagram has been completely assembled.

Chapter 11

Generators and Sequences

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 the 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 array, because there is an infinite amount, 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 where generators may be useful.

Conceptually, a generator is a '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
    }

    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`, and one that is passed an array and initializes the `element` to the array's last valid index.

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

```
let xs = ["A", "B", "C"]

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

> Element 2 of the array is C
> Element 1 of the array is B
> Element 0 of the array is A
> ()
```

Although 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 two (until `NSDecimalNumber` overflows, which is only with extremely large values):

```
class PowerGenerator: GeneratorType {
    typealias Element = NSDecimalNumber

    var power: NSDecimalNumber = NSDecimalNumber(int: 1)
    let two = NSDecimalNumber(int: 2)

    func next() -> Element? {
        power = power.decimalNumberByMultiplyingBy(two)
        return power
    }
}
```

We can use the `PowerGenerator` to inspect increasingly large array indices, for example, when implementing an exponential search algorithm that 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 `NSDecimalNumber -> Bool` as argument, and returns the smallest power of two that satisfies this predicate:

```
func findPower(predicate: NSDecimalNumber -> Bool)
              -> NSDecimalNumber {
    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 1,000:

```
findPower { $0.integerValue > 1000 }
```

```
> 1024
```

The generators we have seen so far all produce numerical elements, but this need not be the case. We can just as 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(contentsOfFile: filename,
```

```
                                    encoding: NSUTF8StringEncoding,
                                    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 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 generic functions that work for every 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, 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 in finding a value that satisfies the predicate. For that reason, find returns an 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. Here is one simple example of a generator transformer that produces the first 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--
            return generator.next()
        }
        else {
            return nil
        }
    }
}
```

Such a generator may be useful when populating an array of a fixed size, or somehow buffering the elements generated.

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 it 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() {
    println("Index \(i) is \(xs[i])")
}
```

```
> Index 2 is C
> Index 1 is B
> Index 0 is A
> ()
```

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) {
    println("Index \(i) is \(xs[i])")
}

> Index 2 is C
> Index 1 is B
> Index 0 is A
> ()
```

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 {
    println("Element is \(x)")
}

> Element is C
> Element is B
> Element is A
> ()
```

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 instead traverse the sequence to produce an array. Mathematicians may therefore object to calling such operations `map`s, as they fail 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 a simple wrapper around regular sequences:

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

If you need to map or filter sequences that may produce either 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 provide 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 is repeatedly called to generate a smaller value; if this value still fails the test, it is 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]? {
        if (!self.isEmpty) {
```

```
            return Array(dropFirst(self))
        }
        return nil
    }
}
```

While this will certainly help shrink counterexamples in *some* cases, there are many different ways to shrink an array. Computing all possible subarrays is an expensive operation. For an array of length `n`, there are `2^n` possible subarrays 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 in which each array is 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
            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.

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]>, whereas 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]))
```

```
> [[2, 3], [1, 3], [1, 2]]
```

A More Functional Approach

Before we refine the removeElement function further, we will rewrite it in a more functional way. The implementation of removeElement that we gave above uses some explicit copying of arrays and mutable state. We can reuse the decompose function from the chapter on Purely Functional Data Structures. Before we do that, we will first rewrite some functions that we already know. 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.decompose {
```

```
        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`. However, 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.decompose {
        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 subarrays as follows:
    - tail of the array is a subarray
    - if we prepend `head` to all the subarrays of the tail, we can compute the subarrays 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.decompose {
        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]))
```

```
> [[2, 3], [1, 3], [1, 2]]
```

Here, 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.decompose {
        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]))
```

```
> [[2, 3], [1, 3], [1, 2], [1, 2, 2], [1, 1, 3], [0, 2, 3]]
```

In addition to generating sublists, this new version of the `smaller` function also produces arrays, where the values of the elements are 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? A first attempt at doing so might result in the following definition:

```
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])
print("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 surprisingly 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 passing 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.

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

Chapter 12

Parser Combinators

Parsers are very useful tools: 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 (2009), 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 lives 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 two 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.

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

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

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.decompose {
            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.decompose {
            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.decompose {
            if condition(head) {
                return one((head, tail))
            }
        }
        return none()
    }
}
```

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

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

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(token(a) <|> token(b), "bcd")

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

Sequence

To combine two parsers that occur 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(token(x), token(y))
testParser(p, "xyz")
```

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

Refining Sequences

The `sequence` function we wrote above is a first 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(token(x), token(y)), token(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(token(x), token(y), token(z))
testParser(p3, "xyz")

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

This returns the expected result, but the 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, and 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 following 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(), token(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 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), token(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 through the combination process multiple times, 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), token(three)),
                      token(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) <*> token(three) <*> token(three), "33")
```

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

Notice that we have defined the `<*>` operator to have left precedence. This means that the operator will first be 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 of how we can now use this operator is to create a parser that combines several characters into a string:

```
let aOrB = token(a) <|> token(b)

func combine(a: Character)(b: Character)(c: Character) -> String {
    return string([a, b, c])
}
let parser = pure(combine) <*> aOrB <*> aOrB <*> token(b)
testParser(parser, "abb")
```

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

In Chapter 3, we defined the `curry` function, which curries a function with two parameters. We can define multiple versions of the `curry` function, 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 { string([$0, $1, $2]) }) <*> aOrB
              <*> aOrB <*> token(b)
testParser(parser2, "abb")
```

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

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

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 `zeroOrMore` function, which executes a parser zero or more times:

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

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

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

Luckily, we can use auto-closures to defer the evaluation of the recursive call to `zeroOrMore` until it is really needed, and with that, break the infinite recursion. To do that, we will first define a helper function, `lazy`. 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 lazy<Token, A>(f: @autoclosure () -> Parser<Token, A>)
        -> Parser<Token, A> {

    return Parser { x in f().p(x) }
}
```

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

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

Let's test the zeroOrMore 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(zeroOrMore(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 `oneOrMore`, which parses something one or more times. It is defined using the `zeroOrMore` combinator:

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

If we parse one or more digits, we get back an array of digits in the form of `Character`s. To convert this into an integer, we can first convert the array of `Character`s 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()! }
             <*> oneOrMore(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`:

```
infix operator </> { precedence 170 }
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 <*> token(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:

```
infix operator <*  { associativity left precedence 150 }
func <* <Token, A, B>(p: Parser<Token, A>,
                      q: Parser<Token, B>) -> Parser<Token, A> {

    return {x in {_ in x} } </> p <*> q
}

infix operator  *> { associativity left precedence 150 }
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 <* token(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,
               operand: Calculator) -> Calculator {

   return curry { evaluate($0, $1) } </> operand
          <* token(character) <*> operand
}

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 parse either an expression of the form `operand operator operand`, or expressions consisting of a single `operand`:

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

   let withOperator = curry { evaluate($0, $1) } </> operand
                         <* token(character) <*> operand
   return withOperator <|> operand
}
```

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 (which consists of a `"+"` expression, followed by a `"-"` character, and then another `"+"` expression) is tried. 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 the 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 the 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 `operand` calculator, optionally followed by the operator and another `operand` 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,
        operand: Calculator) -> Calculator {

    let withOperator = curry(flip(evaluate)) </ token(character)
                             <*> operand
    return optionallyFollowed(operand, 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 chapter, we'll use this parsing library to build a small spreadsheet application.

The calculator we built in this example still has significant shortcomings, the most significant being the fact that you can only use each operator once. In the spreadsheet example we're going to build in the next chapter, we'll remedy this issue.

Chapter 13

Case Study: Building a Spreadsheet Application

In this chapter, we'll build a parser, an evaluator, and a GUI for a very simple spreadsheet application. A spreadsheet consists of cells which are organized in rows and columns. Each cell can contain a formula, for example `10*2`. As a result of parsing the formula, we construct an *abstract syntax tree*, which is a tree structure that describes the formula. Then we evaluate those formula syntax trees to compute the results for each cell, and show them in a table view.

The rows in our spreadsheet are numbered (starting from `0`), and the columns are named (from `A` to `Z`). The expression `C10` refers to the cell in row 10 and column C. An expression in the form of `A0:A10` refers to a list of cells. In this case, it is the list of cells starting with the first cell in column A up to (and including) cell 10 in column A.

To build this spreadsheet application, we'll make use of the parsing library we have built in the previous chapter. We've already used it there to build a simple calculator. Now we'll expand on that to parse the slightly more complicated formulas in the spreadsheet: in addition to simple arithmetic operations, they also support references to other cells and functions, like `SUM(A0:A10)`.

Finally, with the functional core of parsing and evaluation in place, we will show how to integrate this pure functional core with the user interface that's written in an object-oriented way using the standard Cocoa frameworks.

Sample Code

Contrary to many of the other chapters, this chapter comes with a sample project instead of a playground, since the project comes with a simple GUI, and a playground would have been over-challenged with the scope of this example anyway. Please open the "Spreadsheet" project to see the whole example in action.

Parsing

We will divide the parsing phase into two steps: tokenization and parsing. The tokenization step (also called lexing or lexical analysis) transforms the input string into a sequence of tokens. In this process, we also deal with removing whitespace, as well as parsing operator symbols and parentheses, so that we do not have to worry about that in our parser.

The second step — parsing — then operates on the sequence of tokens returned by the tokenizer, and transforms those into an abstract syntax tree: a tree structure that represents a formula expression.

Tokenization

To produce the list of tokens, we could use the `NSScanner` class that comes with Apple's Foundation framework. If we want a nice Swift API, we'd have to wrap it first. In addition, we would need to turn off automatic skipping of whitespace, and handle whitespace ourselves. Therefore, it is much easier and clearer to not use `NSScanner`, and instead write a scanner with the parsing library that we have built.

As we do so often when approaching a problem functionally, we first define the `Token` data type. It's an enum with five cases: numbers, operators, references (such as `A10` or `C42`), punctuation (for now, only parentheses), and function names (such as `SUM` or `AVG`):

```
enum Token: Equatable {
    case Number(Int)
    case Operator(String)
    case Reference(String, Int)
    case Punctuation(String)
    case FunctionName(String)
}
```

Now, for each of those cases we'll define parsers, i.e. functions that consume characters from the input string and return a tuple consisting of a token and the remainder of the input string. For example, assuming the formula string `10+2`, the parser function for the number case would return the tuple `(Token.Number(10), "+2")`.

But before we jump into this, let's first define a couple of helper functions that will make this code clearer. They might not make too much sense on their own, but please bear with us. You'll see them put to good use in just a moment.

The first of those helper functions is called `const` and is pretty straightforward:

```
func const<A, B>(x: A) -> (y: B) -> A {
    return { _ in x }
}
```

You pass a value of type `A` into `const` and it returns a constant function, `B -> A`, i.e. no matter what you pass into the returned function, it will always return what you passed into `const` in the first place.

Another useful helper is the `tokens` function. You pass an array of elements into it, and it constructs a parser that will consume exactly those elements from its input and return a tuple consisting of the array you passed in and the remainder of elements left:

```
func tokens<A: Equatable>(input: [A]) -> Parser<A, [A]> {
    if let (head, tail) = input.decompose {
        return prepend </> token(head) <*> tokens(tail)
    } else {
        return pure([])
    }
}
```

If this looks like magic to you, please make sure to read the parser combinators chapter, which explains all the basic building blocks of the parsing library, like the </> and <*> operators.

Using tokens works recursively on the array you pass in: it splits it into its head (the first element) and tail (the remaining elements), parsing the head element using the token function we already used in the previous chapter, combined with a recursive call to the tokens function itself on the tail.

With this function in place, we can very easily create a string function that constructs a parser that parses a specific string:

```
func string(string: String) -> Parser<Character, String> {
    return const(string) </> tokens(string.characters)
}
```

Finally, we'll introduce a oneOf helper function that lets us combine multiple parsers in a mutually exclusive manner (for example, we want to parse one of the operators +, -, /, and *):

```
func oneOf<Token, A>(parsers: [Parser<Token, A>])
        -> Parser<Token, A> {

    return parsers.reduce(fail(), combine: <|>)
}
```

With all these helpers in place, let's start with the number case. For this, we define a parser that parses natural numbers by consuming one or more decimal digits from the stream of characters and then combines them into an integer, leveraging the power of the reduce function:

```
let pDigit = oneOf(Array(0...9).map { const($0) </> string("\($0)") })

func toNaturalNumber(digits: [Int]) -> Int {
    return digits.reduce(0) { $0 * 10 + $1 }
}

let naturalNumber = toNaturalNumber </> oneOrMore(pDigit)
```

Note how we have already leveraged the `oneOf`, `const`, and `string` helpers to define the natural number parsers in just a few lines of code.

Now we can define a `tNumber` function that simply parses a natural number and then wraps it in the `Number` case to produce a `Token`:

```
let tNumber = { Token.Number($0) } </> naturalNumber
```

To test the number tokenizer, let's run it on the input `"42"`:

```
parse(tNumber, "42")
```

```
> Optional(42)
```

Next up are operators: we have to parse the string representing the operator and wrap it in the `Token`'s `.Operator` case. For each operator that's defined in the array below, we use the `string` function to convert it into a parser, and then combine those parsers for individual operators using the `oneOf` function:

```
let operatorParsers = ["*", "/", "+", "-", ":"].map { string($0) }
let tOperator = { Token.Operator($0) } </> oneOf (operatorParsers)
```

For references, we have to parse a single capital letter followed by a natural number. First we build a `capital` parser using the `charactersFromSet` helper function from the previous chapter:

```
let capitalSet = NSCharacterSet.uppercaseLetterCharacterSet()
let capital = characterFromSet(capitalSet)
```

Now we can use the `capital` parser in conjunction with the `naturalNumber` parser to parse a reference. Since we combine two values, we have to curry the function that constructs the reference:

```
let tReference = curry { Token.Reference(String($0), $1) }
                 </> capital <*> naturalNumber
```

Note that we have to wrap the result of the capital parser in a `String`, since it returns a single character.

The punctuation case is very simple as well: it's either an opening parenthesis or a closing parenthesis, which is then wrapped in the `Punctuation` case:

```
let punctuationParsers = ["(", ")"].map { string($0) }
let tPunctuation = { Token.Punctuation($0) }
                   </> oneOf(punctuationParsers)
```

Finally, function names (such as `SUM`) consist of one or more capital letters, which are converted into a string and wrapped in the `FunctionName` case:

```
let tName = { Token.FunctionName(String($0)) }
            </> oneOrMore(capital)
```

Now we have in place all the functions that we need to generate a stream of tokens for our formula expressions. Since we want to ignore any kind of whitespace in those formulas, we define one more helper function, `ignoreLeadingWhitespace`, that 'eats up' whitespace between tokens:

```
let whitespaceSet = NSCharacterSet.whitespaceAndNewlineCharacterSet()
let whitespace = characterFromSet(whitespaceSet)

func ignoreLeadingWhitespace<A>(p: Parser<Character, A>)
        -> Parser<Character, A> {

    return zeroOrMore(whitespace) *> p
}
```

The complete expression parser can now be defined as a combination of all the parsers above with the `oneOf` function, wrapped with the `ignoreLeadingWhitespace` function to dispose of whitespace, and finally wrapped with the `zeroOrMore` function to make sure we get a list of tokens back:

```
func tokenize() -> Parser<Character, [Token]> {
    let tokenParsers = [tNumber, tOperator, tReference,
                        tPunctuation, tName]
    return zeroOrMore(ignoreLeadingWhitespace(oneOf(tokenParsers)))
}
```

That's all there is to the tokenizer. Now we can run it on a sample expression:

```
parse(tokenize(), "1+2*3+SUM(A4:A6)")
```

```
> Optional([1, +, 2, *, 3, +, SUM, (, A4, :, A6, )])
```

Parsing

From the list of tokens generated by our tokenizer, we now create an expression. Expressions can be either numbers, references (to another cell), binary expressions with an operator, or function calls. We can capture that in the following recursive enum, which is the abstract syntax tree for spreadsheet expressions:

```
protocol ExpressionLike {
    func toExpression() -> Expression
}

enum Expression {
    case Number(Int)
    case Reference(String, Int)
    case BinaryExpression(String, ExpressionLike, ExpressionLike)
```

```
    case FunctionCall(String, ExpressionLike)
}

extension Expression: ExpressionLike {
    func toExpression() -> Expression {
        return self
    }
}
```

Defining the protocol `ExpressionLike` is a workaround to be able to define a recursive enum, since Swift currently doesn't allow true recursive enums.

So far, we have only constructed parsers that work on strings (or to be more precise, a list of characters). However, we have defined our parsers in such a way that they can work on any type of tokens, including the `Token` type we have defined above.

First, we define a simple type alias for our parser, which says that it computes `Expression` values out of a stream of `Token` values:

```
typealias ExpressionParser = Parser<Token, Expression>
```

Let's start with parsing numbers. When trying to parse a token to a number expression, two things can happen: either the parsing succeeds in case the token was a number token, or it fails for all other cases. To construct our number parser, we'll define a helper function, `optionalTransform`, that lets us transform the parsed token to an expression, or return `nil` in case the token can't be transformed into a number expression:

```
func optionalTransform<A, T>(f: T -> A?) -> Parser<T, A> {
    return { f($0)! } </> satisfy { f($0) != nil }
}
```

Now we can define the number parser. Note that the `Number` case is used twice: the first instance is the `Token.Number` case, and the second instance is the `Expression.Number` case:

```
let pNumber: ExpressionParser = optionalTransform {
    switch $0 {
```

```
        case .Number(let number):
            return Expression.Number(number)
        default:
            return nil
    }
}
```

Parsing references is equally simple:

```
let pReference: ExpressionParser = optionalTransform {
    switch $0 {
        case .Reference(let column, let row):
            return Expression.Reference(column, row)
        default:
            return nil
    }
}
```

Finally, we combine those two into one parser:

```
let pNumberOrReference = pNumber <|> pReference
```

We can now apply this parser to both numbers and references to see that it works as expected:

```
parse(pNumberOrReference, parse(tokenize(), "42")!)

> Optional(42)

parse(pNumberOrReference, parse(tokenize(), "A5")!)

> Optional(A5)
```

Next, let's take a look at function calls such as `SUM(...)`. In order to do that, we first define a parser, similar to the number and reference parsers above, that parses a function name token or returns `nil` if the token is not of type `Token.FunctionName`:

```
let pFunctionName: Parser<Token, String> = optionalTransform {
    switch $0 {
        case .FunctionName(let name):
            return name
        default:
            return nil
    }
}
```

In our case, the argument of a function call is always a list of cell references in the form of `A1:A3`. The list parser has to parse two reference tokens separated by a `:` operator token:

```
func makeList(l: Expression, r: Expression) -> Expression {
    return Expression.BinaryExpression(":", l, r)
}

let pList: ExpressionParser = curry(makeList) </> pReference
                              <* op(":") <*> pReference
```

`op` is a simple helper that takes an operator string and creates a parser that parses a corresponding operator token, returning the operator string as a result.

Before we can put everything together to parse a function call, we still need a function to parse a parenthesized expression:

```
func parenthesized<A>(p: Parser<Token, A>) -> Parser<Token, A> {
    return token(Token.Punctuation("(")) *> p
           <* token(Token.Punctuation(")"))
}
```

This function takes any parser `p` as argument, and combines it with a parser for an opening parenthesis on the left, and one for a closing parenthesis on the right. The result of `parenthesized` is the same as the result of parsing `p`, since the parentheses get thrown away due to the use of the `*>` and `<*` combinator operators.

Now we can combine all those elements:

```
func makeFunctionCall(name: String, arg: Expression) -> Expression {
    return Expression.FunctionCall(name, arg)
}
let pFunctionCall = curry(makeFunctionCall) </> pFunctionName
                    <*> parenthesized(pList)
```

Let's give it a try:

```
parse(pFunctionCall, parse(tokenize(), "SUM(A1:A3)")!)
```

```
> Optional(SUM((A1) : (A3)))
```

To parse whole formula expressions, we have to start with the smallest building blocks that expressions consist of, and work our way outward to respect operator precedence. For example, we want to parse multiplications with higher precedence than additions. To start with, we define a parser for formula primitives:

```
let pParenthesizedExpression = parenthesized(lazy(expression()))
let pPrimitive = pNumberOrReference <|> pFunctionCall
                 <|> pParenthesizedExpression
```

A primitive is either a number, a reference, a function call, or another expression wrapped in brackets. For the latter, we need to use the `lazy` helper function, which ensures that `expressionParser` only gets called if it really needs to be, otherwise we would get stuck in an endless loop.

Now that we can parse primitives, we'll put them together to parse products (or divisions for that matter — we'll treat them as equal, as they have the same operator precedence). A product consists of at least one factor, followed by zero or more `* factor` or `/ factor` pairs. Those pairs can be modeled as:

```
let pMultiplier = curry { ($0, $1) } </> (op("*") <|> op("/"))
                  <*> pPrimitive
```

The result of the `pMultiplier` parser is tuples consisting of the operator string (e.g. `"+"`) and the expression to their right. Consequently, the whole product parser looks like this:

```
let pProduct = curry(combineOperands) </> pPrimitive <*>
               zeroOrMore(pMultiplier)
```

The key here is the `combineOperands` function, which builds an expression tree from one primitive and zero or more multiplier tuples. We have to make sure to respect the operator's left-associativity when building this tree (it doesn't matter for `*`, since it is commutative, but it does for `/`). Luckily, `reduce` works in a left-associative manner, i.e. a sequence like `1, 2, 3, 4` is reduced in the order `(((1, 2), 3), 4)`. We leverage this behavior to combine multiple operands into one expression:

```
func combineOperands(first: Expression,
                     rest: [(String, Expression)]) -> Expression {

    return rest.reduce(first, combine: { result, pair in
        let (op, exp) = pair
        return Expression.BinaryExpression(op, result, exp)
    })
}
```

Now we can parse primitives and products of primitives. The last missing piece in the puzzle is the addition of one or more of those products. Since this follows exactly the same pattern as the product itself, we'll keep it brief:

```
let pSummand = curry { ($0, $1) } </> (op("-")
               <|> op("+")) <*> pProduct
let pSum = curry(combineOperands) </> pProduct
           <*> zeroOrMore(pSummand)
```

A whole formula expression can now be parsed by the `pSum` parser. We alias it as `expression`:

```
expression = { pSum }
```

Now we combine our tokenizer and parser into a function, `parseExpression`. It first tokenizes the input string, and then — if this succeeds — parses the expression:

```
func parseExpression(input: String) -> Expression? {
    if let tokens = parse(tokenize(), input) {
        return parse(expression(), tokens)
    }
    return nil
}
```

Using `flatMap` for optionals, we simplify this function as follows:

```
func parseExpression(input: String) -> Expression? {
    return flatMap(parse(tokenize(), input)) {
        parse(expression(), $0)
    }
}
```

Applying this function to a complete formula yields the expression tree as we would expect:

```
parseExpression("1 + 2*3 - MIN(A5:A9)")
```

```
> Optional(((1) + ((2) * (3))) - (MIN((A5) : (A9))))
```

Evaluation

Now that we have an abstract syntax tree (in the form of `Expression` values), we can evaluate these expressions into results. For our simple example, we will assume a one-dimensional spreadsheet (i.e. with only one column). It's not too hard to extend the example into a two-dimensional spreadsheet, but for explanatory purposes, sticking to one dimension makes things clearer.

In order to make the code easier to read, we'll make another simplification: we don't cache the results of evaluating cell expressions. In a 'real' spreadsheet, it would be important to define the order of operations. For example, if cell `A2` uses the value of `A1`, it is useful to first compute `A1` and cache that result, so that we can use the cached result when computing

A2's value. Adding this is not hard, but we leave it as an exercise for the reader for the sake of clarity.

That said, let's start by defining our `Result` enum. When we evaluate an expression, there are three different possible results. In the case of simple arithmetic expressions such as `10+10`, the result will be a number, which is covered by the case `IntResult`. In the case of an expression like `A0:A10`, the result is a list of results, which is covered by the case `ListResult`. Finally, it is very possible that there is an error, for example, when an expression can't be parsed, or when an illegal expression is used. In this case, we want to display an error, which is covered by the `EvaluationError` case:

```
enum Result {
    case IntResult(Int)
    case ListResult([Result])
    case EvaluationError(String)
}
```

Before we continue with the evaluation, we first define a helper function called `lift`. It makes it possible for us to use functions that normally work on integers to work on the `Result` enum. For example, the + operator is a function that takes two integers and sums them up. Using `lift`, we can lift the + operator into a function that takes two `Result` values, and if both are `IntResult`s, it combines them into a new `IntResult`. If either of the `Result` values is not an `IntResult`, the function returns an evaluation error:

```
func lift(f: (Int, Int) -> Int) -> ((Result, Result) -> Result) {
    return { l, r in
        switch (l, r) {
            case (.IntResult(let x), .IntResult(let y)):
                return .IntResult(f(x, y))
            default:
                return .EvaluationError("Type error, " +
                        "couldn't evaluate \(l) + \(r)")
        }
```

```
    }
}
```

Our tokenizer supports the "+", "/", "*", and "-" operators. However, in our expression enum, there's the BinaryExpression case, which stores the operator as a String value. Therefore, we need a way to map those strings into functions that combine two Ints. We define a dictionary, integerOperators, which captures exactly that. Note that we have to wrap every operator with the function o to convince the compiler, though hopefully this will not be necessary in future releases of the language:

```
func o(f: (Int, Int) -> Int) -> (Int, Int) -> Int {
    return f
}

let integerOperators: Dictionary<String, (Int, Int) -> Int> =
    [ "+": o(+), "/": o(/), "*": o(*), "-": o(-) ]
```

Now we can write a function, evaluateIntegerOperator, that, given an operator string op and two expressions, returns a result. As an additional parameter, it gets a function, evaluate, which knows how to evaluate an expression. The operator op is looked up in the integerOperators dictionary, which returns an optional function with type (Int, Int) -> Int. We use the map on optionals, and then evaluate the left argument and the right argument, giving us two results. We finally use the lift function to combine the two results into a new result. Note that if the operator couldn't be recognized, this returns nil:

```
func evaluateIntegerOperator(op: String,
                             l: Expression,
                             r: Expression,
                             evaluate: Expression? -> Result)
                             -> Result? {

    return integerOperators[op].map {
        lift($0)(evaluate(l), evaluate(r))
```

```
    }
}
```

To evaluate the list operator (e.g. `A1:A5`), we first check if the operator string is actually the list operator `:`. Also, because we only support one column, we want to make sure that both `l` and `r` are references, and that `row2` is larger than or equal to `row1`. Note that we can check all of those conditions at the same time using a single switch case; in all other cases, we simply return `nil`. In the case that all our preconditions are fulfilled, we map over each cell, and evaluate the result:

```
func evaluateListOperator(op: String,
                          l: Expression,
                          r: Expression,
                          evaluate: Expression? -> Result)
                          -> Result? {

    switch (op, l, r) {
        case (":", .Reference("A", let row1),
                   .Reference("A", let row2)) where row1 <= row2:
            return Result.ListResult(Array(row1...row2).map {
                evaluate(Expression.Reference("A", $0))
            })
        default:
            return nil
    }
}
```

Now we're ready to evaluate any binary operator. First, we try if the integer operator succeeds. In case this returns `nil`, we try to evaluate the list operator. In case that also doesn't succeed, we return an error:

```
func evaluateBinary(op: String,
                    l: Expression,
                    r: Expression,
                    evaluate: Expression? -> Result) -> Result {
```

```
    return evaluateIntegerOperator(op, l, r, evaluate)
           ?? evaluateListOperator(op, l, r, evaluate)
           ?? .EvaluationError("Couldn't find operator \(op)")
}
```

For now, we'll support two functions on lists, `SUM` and `MIN`, which compute the sum and the minimum of a list, respectively. Given a function name and the result of the parameter (we currently only support functions with exactly one parameter), we switch on the function name and check in both cases if the parameter is a list result. If yes, we simply compute the sum or the minimum of the results. Because these lists are not lists with `Int`s, but rather lists with `Result`s, we need to lift the operator using the `lift` function:

```
func evaluateFunction(functionName: String,
                      parameter: Result) -> Result {

    switch (functionName, parameter) {
        case ("SUM", .ListResult(let list)):
            return list.reduce(Result.IntResult(0), lift(+))
        case ("MIN", .ListResult(let list)):
            return list.reduce(Result.IntResult(Int.max),
                               lift { min($0, $1) })
        default:
            return .EvaluationError("Couldn't evaluate function")
    }
}
```

We finally have all the pieces together to evaluate a single expression. In order to do this, we also need all other expressions (because the expression might reference another cell). Therefore, the `evaluateExpression` function takes a `context` argument that is an array of optional expressions: one for each cell (the first element in the array is `A0`, the second `A1`, and so on). In case parsing failed, the optional expression has a `nil` value.

All that `evaluateExpression` does is switch on the expression and call the appropriate evaluation functions:

```
func evaluateExpression(context: [Expression?])
        -> Expression? -> Result {

    return { (e: Expression?) in
        e.map { expression in
            let recurse = evaluateExpression(context)
            switch (expression) {
                case .Number(let x):
                    return Result.IntResult(x)
                case .Reference("A", let idx):
                    return recurse(context[idx])
                case .BinaryExpression(let s, let l, let r):
                    return evaluateBinary(s, l.toExpression(),
                                          r.toExpression(), recurse)
                case .FunctionCall(let f, let p):
                    return evaluateFunction(f,
                                            recurse(p.toExpression()))
                default:
                    return .EvaluationError("Couldn't evaluate " +
                                            "expression")
            }
        } ?? .EvaluationError("Couldn't parse expression")
    }
}
```

Note that, in case of a reference, we look up the referenced cell in the context and recursively call ourselves. If the spreadsheet accidentally contains a circular reference (e.g. A0 depends on A1, and A1 depends on A0), this function will call itself recursively until a stack overflow occurs.

Finally, we can define a convenience function, `evaluateExpressions`, which takes a list of optional expressions, and produces a list of results by mapping `evaluateExpression` over it:

```
func evaluateExpressions(expressions: [Expression?]) -> [Result] {
    return expressions.map(evaluateExpression(expressions))
}
```

As stated in the introduction of this chapter and throughout the text, there are a lot of limitations to the current parser and evaluator. There could be better error messages, a dependency analysis of the cells, loop detection, massive optimizations, and much more. But note that we have defined the entire model layer of a spreadsheet, parsing, and evaluation in about 200 lines. This is the power of functional programming: after thinking about the types and data structures, many of the functions are fairly straightforward to write. And once the code is written, we can always make it better and faster.

Furthermore, we were able to reuse many of the standard library functions within our expression language. The `lift` combinator made it really simple to reuse any binary function that works on integers. We were able to compute the results of functions like `SUM` and `MIN` using `reduce`. And with the parsing library from the previous chapter in place, we quickly wrote both a lexer and a parser.

The GUI

Now that we have the formula parser and evaluator in place, we can build a simple GUI around them. Because almost all frameworks in Cocoa are object oriented, we'll have to mix OOP and functional programming to get this working. Luckily, Swift makes that really easy.

We will create an XIB containing our window, which contains a single `NSTableView` with one column. The table view is populated by a data source, and has a delegate to handle the editing. Both the data source and delegate are separate objects. The data source stores the formulas and their results, and the delegate tells the data source which row is currently being edited. The edited row is then displayed as a formula rather than a result.

The Data Source

Our data source is exactly where the functional and OOP parts mix. It is an object that conforms to the `NSTableViewDataSource` protocol, which provides the data for an `NSTableView`. In our case, the data is the cells. Depending on whether a cell is currently being edited, we display either the formula (when edited), or the result. In our class, we need to store the formulas, the computed results, and the row that is currently being edited (which might be `nil`):

```
class SpreadsheetDatasource: NSObject, NSTableViewDataSource {
    var formulas: [String]
    var results: [Result]
    var editedRow: Int? = nil
```

In the initializer, we initialize the formulas with the string `"0"`, and the initial values with the corresponding result (zero):

```
    override init() {
        let initialValues = Array(1..<10)
        formulas = initialValues.map { _ in "0" }
        results = initialValues.map { Result.IntResult($0) }
    }
```

The table view's delegate protocol requires us to implement two methods: the first method returns the number of rows in the table view (which is simply the number of formulas). The second method returns the content for each cell. In our case, we look at the row, and if it's the row that's currently being edited, we return the formula. Otherwise, we return the result:

```
    func numberOfRowsInTableView(tableView: NSTableView!) -> Int {
        return formulas.count
    }

    func tableView(aTableView: NSTableView!,
                   objectValueForTableColumn: NSTableColumn!,
```

```
                    row: Int) -> AnyObject! {

        return editedRow == row ? formulas[row]
                                 : results[row].description
    }
```

When the user has edited a cell, we reevaluate all formulas:

```
    func tableView(aTableView: NSTableView!,
                   setObjectValue: AnyObject!,
                   forTableColumn: NSTableColumn!,
                   row: Int) {

        let string = setObjectValue as String
        formulas[row] = string
        calculateExpressions()
        aTableView.reloadData()
    }
```

In order to calculate expressions, we map over each formula to tokenize and parse it. We use the `parseExpression` function defined earlier. This will return a list of optional expressions (in case either tokenization or parsing has failed). Then, we use the `evaluate` function to compute a list of results. Because of key-value observing, the GUI will update automatically:

```
    func calculateExpressions() {
        let expressions: [Expression?] = formulas.map(parseExpression)
        results = evaluate(expressions)
    }
}
```

The Delegate

The delegate's only task is to let the data source know which cell is currently being edited. Because we'd like a loose coupling between the two, we'll introduce an extra protocol, `EditedRow`, which describes that there should be a property, `editedRow`, which optionally contains the value of the row that's currently being edited:

```
protocol EditedRow {
    var editedRow: Int? { get set }
}
```

Now, when the row is edited, we just set that property on the data source and we're done:

```
class SpreadsheetDelegate: NSObject, NSTableViewDelegate {
    var editedRowDelegate: EditedRow?

    func tableView(aTableView: NSTableView!,
                   shouldEditTableColumn aTableColumn: NSTableColumn!,
                   row rowIndex: Int) -> Bool {

        editedRowDelegate?.editedRow = rowIndex
        return true
    }
}
```

The Window Controller

The final bit that ties everything together is the window controller. (If you come from iOS, this fulfills a similar role as `UIViewController` does.) The window controller has outlets to the table view, the data source, and the delegate. All of these objects are instantiated by the nib file:

```
class SheetWindowController: NSWindowController {

    @IBOutlet var tableView: NSTableView! = nil
    @IBOutlet var dataSource: SpreadsheetDatasource?
    @IBOutlet var delegate: SpreadsheetDelegate?
```

When the window loads, we hook up the delegate with the data source so that it can notify the data source about edited rows. Also, we register an observer that will let us know when editing ends. When the notification for the end of editing is fired, we set the data source's edited row to `nil`:

```
    override func windowDidLoad()  {
        delegate.editedRowDelegate = dataSource

        NSNotificationCenter.defaultCenter().addObserver(
          self,
          selector: NSSelectorFromString("endEditing:"),
          name: NSControlTextDidEndEditingNotification,
          object: nil)
    }

    func endEditing(note: NSNotification) {
        if note.object as NSObject === tableView {
            dataSource.editedRow = nil
        }
    }
}
```

This is all there is to it. We now have a fully working prototype of a working, single-column spreadsheet. Check out the sample project for the complete example.

Chapter 14

Functors, Applicative Functors, and Monads

In this chapter, we will explain some terminology and common patterns used in functional programming, including functors, applicative functors, and monads. Understanding these common patterns will help you design your own data types and choose the correct functions to provide in your APIs.

Functors

Thus far, we have seen two different functions named `map` with the following types:

```
func map<T, U>(xs: [T], transform: T -> U) -> [U]
```

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

Why have two such different functions with the same name? To answer that question, let's investigate how these functions are related. To begin with, 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 of the `map` function on arrays and optionals, the similarity becomes more apparent:

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

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

Both `Optional` and `Array` are *type constructors* that expect a generic type argument. For instance, `Array<T>` and `Optional<Int>` are valid types, but `Array` by itself is not. Both of these `map` functions take two arguments: the structure being mapped, and a function `transform` of type `T -> U`. The `map` functions use a function argument 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*.

In fact, there are many other types that we have defined that are indeed functors. For example, we can implement a `map` function on the `Box` and `Result` types from Chapter 8:

```
func map<T, U>(box: Box<T>, transform: T -> U) -> Box<U> {
    return Box(transform(box.value))
}

func map<T, U> (result: Result<T>, transform: T -> U) -> Result<U> {
    switch result {
        case let Result.Success(box):
            return Result.Success(map(box, transform))
        case let Result.Failure(error):
            return Result.Failure(error)
    }
}
```

Similarly, the types we have seen for binary search trees, tries, and parser combinators are all functors. Functors are sometimes described as 'containers' storing values of some type. The `map` functions transform the stored values stored in a container. This can be a useful intuition, but it can be too restrictive. Remember the `Region` type that we saw in Chapter 2?

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

Using this definition of regions, we can only generate black and white bitmaps. We can generalize this to abstract over the kind of information we associate with every position:

```
struct Region<T> {
    let value : Position -> T
}
```

Note that we need to introduce a struct here, as we cannot define generic type aliases in Swift. Using this definition, we can associate booleans, RGB values, or any other information with every position. We can also define a `map` function on these generic regions. Essentially, this definition boils down to function composition:

```
func map<T, U>(region: Region<T>, transform: T -> U) -> Region<U> {
    return Region { pos in transform(region.value(pos)) }
}
```

Such regions are a good example of a functor that does not fit well with the intuition of functors being containers. Here, we have represented regions as *functions*, which seem very different from containers.

Almost every generic enumeration that you can define in Swift will be a functor. Providing a `map` function gives fellow developers a powerful, yet familiar, function for working with such enumerations.

Applicative Functors

Many functors also support other operations aside from `map`. For example, the parsers from Chapter 12 were not only functors, but also defined the following two operations:

```
func pure<Token, A>(value: A) -> Parser<Token, A>

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

The `pure` function explains how to turn any value into a (trivial) parser that returns that value. Meanwhile, the `<*>` operator sequences two parsers: the first parser returns a function, and the second parser returns an argument for this function. The choice for these two operations is no coincidence. Any type constructor for which we can define appropriate `pure` and `<*>` operations is called an *applicative functor*. To be more precise, a functor `F` is applicative when it supports the following operations:

```
func pure<A>(value: A) -> F<A>

func <*><A,B>(f: F<A -> B>, x: F<A>) -> F<B>
```

Applicative functors have been lurking in the background throughout this book. For example, the `Region` struct defined above is also an applicative functor:

```
func pure<A>(value: A) -> Region<A> {
    return Region.Region { pos in value }
}

func <*><A,B>(regionF: Region<A -> B>,
              regionX: Region<A>) -> Region<B> {

    return Region { pos in regionF.value(pos)(regionX.value(pos)) }
}
```

Now the `pure` function always returns a constant value for every region. The `<*>` operator distributes the position to both its region arguments, which yields a function of type `A -> B`, and a value of type `A`. It then combines these in the obvious manner, by applying the resulting function to the argument.

Many of the functions defined on regions can be described succinctly using these two basic building blocks. Here are a few example functions — inspired by Chapter 1 — written in applicative style:

```
func everywhere() -> Region<Bool> {
    return pure(true)
}

func invert(region: Region<Bool>) {
    return pure(!) <*> region
}

func intersection(region1: Region<Bool>, region2: Region<Bool>) {
    return pure(&&) <*> region1 <*> region2
}
```

This shows how the applicative instance for the `Region` type can be used to define pointwise operations on regions.

Applicative functors are not limited to regions and parsers. Swift's built-in optional type is another example of an applicative functor. The corresponding definitions are fairly straightforward:

```
func pure<A>(value: A) -> A? {
    return value
}

func <*><A, B>(optionalTransform: (A -> B)?,
               optionalValue: A?) -> B? {

    if let transform = optionalTransform {
```

```
        if let value = optionalValue {
            return transform(value)
        }
    }
    return nil
}
```

The `pure` function wraps a value into an optional. This is usually handled implicitly by the Swift compiler, so it's not very useful to define ourselves. The `<*>` operator is more interesting: given a (possibly `nil`) function and a (possibly `nil`) argument, it returns the result of applying the function to the argument when both exist. If either argument is `nil`, the whole function returns `nil`. We can give similar definitions for `pure` and `<*>` for the `Result` type from Chapter 7.

By themselves, these definitions may not be very interesting, so let's revisit some of our previous examples. You may want to recall the `addOptionals` function, which tried to add two possibly `nil` integers:

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

Using the definitions above, we can give a short and sweet alternative definition of `addOptionals`:

```
func addOptionals(maybeX: Int?, maybeY: Int?) -> Int? {
    return pure(curry(+)) <*> maybeX <*> maybeY
}
```

Once you understand the control flow that operators like `<*>` encapsulate, it becomes much easier to assemble complex computations in this fashion.

There is one other example from the optionals chapter that we would like to revisit:

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

Here we consulted one dictionary, `capitals`, to retrieve the capital city of a given country. We then consulted another dictionary, `cities`, to determine each city's population. Despite the obvious similarity to the previous `addOptionals` example, this function *cannot* be written in applicative style. Here is what happens when we try to do so:

```
func populationOfCapital(country: String) -> Int? {
    return { pop in pop * 1000 } <*> capitals[country] <*> cities[...]
}
```

The problem is that the *result* of the first lookup, which was bound to the `capital` variable in the original version, is needed in the second lookup. Using only the applicative operations, we quickly get stuck: there is no way for the result of one applicative computation (`capitals[country]`) to influence another (the lookup in the `cities` dictionary). To deal with this, we need yet another interface.

The M-Word

In Chapter 4, we gave the following alternative definition of `populationOfCapital`:

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

```
            return population * 1000
        }
    }
}
```

Here we used a custom operator, `>>=`, to combine optional computations:

```
infix operator >>= {}

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

How is this different from the applicative interface? The types are subtly different. In the applicative `<*>` operation, *both* arguments are optionals. In the `>>=` operator, on the other hand, the second argument is a *function* that returns an optional value. Consequently, we can pass the result of the first dictionary lookup on to the second.

The `>>=` operator is impossible to define in terms of the applicative functions. In fact, the `>>=` operator is one of the two functions supported by *monads*. More generally, a type constructor `F` is a monad if it defines the following two functions:

```
func pure<A>(value: A) -> F<A>

func >>=<A, B>(x: F<A>, f: A -> F<B>) -> F<B>
```

The `>>=` operator is sometimes pronounced "bind," as it binds the result of the first argument to the parameter of its second argument.

In addition to Swift's optional type, the `Result` enumeration defined in Chapter 7 is also a monad. This insight makes it possible to chain together computations that may return an `NSError`. For example, we could define a function that copies the contents of one file to another as follows:

```
func copyFile(sourcePath: String, targetPath: String,
              encoding: Encoding) -> Result<()> {

    return readFile(sourcePath, encoding) >>= { contents in
        writeFile(contents, targetPath, encoding)
    }
}
```

If the call to either `readFile` or `writeFile` fails, the `NSError` will be logged in the result. This may not be quite as nice as Swift's optional binding mechanism, but it is still pretty close.

There are many other applications of monads aside from handling errors. For example, arrays are also a monad:

```
func pure<A>(value: A) -> [A] {
    return [value]
}

func >>=<A, B>(f: A -> [B], xs: [A]) -> [B] {
    return flatten(xs.map(f))
}
```

The `flatten` function, defined in Chapter 3, flattens an array of arrays into a single array. The `>>=` operator defined here is the same as the `flatMap` function we previously defined on sequences.

What have we gained from these definitions? The monad structure of arrays provides a convenient way to define various combinatorial functions or solve search problems. For example, suppose we need to compute the *cartesian product* of two arrays, `xs` and `ys`. The cartesian product consists of a new array of tuples, where the first component of the tuple is drawn from `xs`, and the second component is drawn from `ys`. Using a for loop directly, we might write:

```
func cartesianProduct1<A, B>(xs: [A], ys: [B]) -> [(A, B)] {
    var result: [(A, B)] = []
    for x in xs {
```

```
        for y in ys {
            result += [(x, y)]
        }
    }
    return result
}
```

We can now rewrite `cartesianProduct` to use the monadic `>>=` operator instead of for loops:

```
func cartesianProduct2<A, B>(xs: [A], ys: [B]) -> [(A, B)] {
    return xs >>= { x in ys >>= { y in [(x, y)] } }
}
```

The `>>=` operator allows us to take an element `x` from the first array, `xs`; next, we take an element `y` from `ys`. For each pair of `x` and `y`, we return the array `[(x, y)]`. The `>>=` operator handles combining all these arrays into one large result.

While this example may seem a bit contrived, the `>>=` operator on arrays has many important applications. Languages like Haskell and Python support special syntactic sugar for defining lists, called *list comprehensions*. These list comprehensions allow you to draw elements from existing lists and check that these elements satisfy certain properties. They can all be desugared into a combination of maps, filters, and `>>=`.

Discussion

Why care about these things? Does it really matter if you know that some type is an applicative functor or a monad? We think it does.

Consider the parser combinators from Chapter 11. Defining the correct way to sequence two parsers is not easy: it requires a bit of insight into how parsers work. Yet it is an absolutely essential piece of our library, without which we could not even write the simplest parsers. If you have the insight that our parsers form an applicative functor, you may realize that the existing `<*>` provides you with exactly the right notion of sequencing two parsers, one after the other. Knowing what abstract operations your types support can help you find such complex definitions.

Abstract notions, like functors, provide important vocabulary. If you ever encounter a function named `map`, you can probably make a pretty good guess as to what it does. Without a precise terminology for common structures like functors, you would have to rediscover each new `map` function from scratch.

These structures give guidance when designing your own API. If you define a generic enumeration or struct, chances are that it supports a `map` operation. Is this something that you want to expose to your users? Is your data structure also an applicative functor? Is it a monad? What do the operations do? Once you familiarize yourself with these abstract structures, you see them pop up again and again.

Although it is harder in Swift than in Haskell, you can define generic functions that work on any applicative functor. Functions such as the `</>` operator on parsers were defined exclusively in terms of the applicative `pure` and `<*>` functions. As a result, we may want to redefine them for *other* applicative functors aside from parsers. In this way, we recognize common patterns in how we program using these abstract structures; these patterns may themselves be useful in a wide variety of settings.

The historical development of monads in the context of functional programming is interesting. Initially, monads were developed in a branch of Mathematics known as *category theory*. The discovery of their relevance to Computer Science is generally attributed to Moggi (1991) and later popularized by Wadler (1992a; 1992b). Since then, they have been used by functional languages such as Haskell to contain side effects and I/O (Peyton Jones 2001). Applicative functors were first described by McBride and Paterson (2008), although there were many examples already known. A complete overview of the relation between many of the abstract concepts described in this chapter can be found in the Typeclassopedia (Yorgey 2009).

Chapter 15

Conclusion

Further Reading

Where to go from here? Because Swift has not been out very long, not many books have been published covering advanced topics. That said, we have tried to compile a list of blogs that we have enjoyed reading in the past months:

- Both Rob Napier and Alexandros Salazar have regularly blogged about all kinds of topics related to functional programming and Swift.

- NSHipster covers some of the more obscure and interesting corners of the Swift language.

- Airspeed Velocity provides one of the most comprehensive overviews of the Swift standard library and its evolution over time.

- And finally, we should, of course, mention objc.io. Despite what the publication's name might suggest, Issue 16 was exclusively about Swift. Furthermore, we expect to publish more articles about Swift as the popularity of the language continues to grow.

One way to further hone your functional programming skills is by learning Haskell. There are many other functional languages, such as F#, OCaml, Standard ML, Scala, or Racket, each of which would make a fine choice of language to complement Swift. Haskell, however, is the most likely to challenge your preconceptions about programming. Learning to program well in Haskell will change the way you work in Swift.

There are a lot of Haskell books and courses available these days. Graham Hutton's *Programming in Haskell* (2007) is a great starting point to familiarize yourself with the language basics. *Learn You a Haskell for Great Good!* is free to read online and covers some more advanced topics. *Real World Haskell* describes several larger case studies and a lot of the technology missing from many other books, including support for profiling, debugging, and automated testing. Richard Bird is famous for his "functional pearls" — elegant, instructive examples of functional programming, many of which can be found in his book, *Pearls of Functional Algorithm Design* (2010), or online. Finally, *The Fun of Programming* is a collection of domain-specific languages, embedded in Haskell, covering domains ranging from financial contracts to hardware design (Gibbons and de Moor 2003). Additionally, if you want hands-on experience in a higher-learning environment, The University of Utrecht organizes an annual Summer School on Applied Functional Programming that caters to Haskell beginners and intermediate programmers alike.

If you want to learn more about programming language design in general, Benjamin Pierce's *Types and Programming Languages* (2002) is an obvious choice. Bob Harper's *Practical Foundations for Programming Languages* (2012) is more recent and more rigorous, but unless you have a solid background in computer science or mathematics, you may find it hard going.

Don't feel obliged to make use of all of these resources; many of them may not be of interest to you. But you should be aware that there is a huge amount of work on programming language design, functional programming, and mathematics that has directly influenced the design of Swift.

What is Functional Programming?

So what is functional programming? Many people (mistakenly) believe functional programming is *only* about programming with higher-order functions, such as `map` and `filter`. There is much more to it than that.

In the Introduction, we mentioned three qualities that we believe characterize well-designed functional programs in Swift: modularity, a careful treatment of mutable state, and types. In each of the chapters we have seen, these three concepts pop up again and again.

Higher-order functions can certainly help define certain abstractions, such as the `Filter` type in Chapter 3, but they are a means, not an end. The functional wrapper around the Core Image library we defined provides a type-safe and modular way to assemble complex image filters. Generators and sequences (Chapter 11) help us abstract iteration.

Swift's advanced type system can help catch many errors before your code is even run. Optional types (Chapter 5) mark possible `nil` values as suspicious; generics not only facilitate code reuse, but also allow you to enforce certain safety properties (Chapter 4); and enumerations and structs provide the building blocks to model the data in your domain accurately (Chapters 8 and 9).

Referentially transparent functions are easier to reason about and test. Our QuickCheck library (Chapter 6) shows how we can use higher-order functions to *generate* random unit tests for referentially transparent functions. Swift's careful treatment of value types (Chapter 7) allows you to share data freely within your application, without having to worry about it changing unintentionally or unexpectedly.

We can use all these ideas in concert to build powerful domain-specific languages. Our libraries for diagrams (Chapter 10) and parser combinators (Chapter 12) both define a small set of functions, providing the modular building blocks that can be used to assemble solutions to large and difficult problems. Our final case study shows how these domain-specific languages can be used in a complete application (Chapter 13).

Finally, many of the types we have seen share similar functions. In Chapter 14 we show how to group them and how they relate to each other.

Closure

This is an exciting time for Swift. The language is still very much in its infancy. Compared to Objective-C, there are many new features — borrowed from existing functional programming languages — that have the potential to dramatically change the way we write software for iOS and OS X.

At the same time, it is unclear how the Swift community will develop. Will people embrace these features? Or will they write the same code in Swift as they do in Objective-C, but without the semicolons? Time will tell. By writing this book, we hope to have introduced you to some concepts from functional programming. It is up to you to put these ideas in practice as we continue to shape the future of Swift.

Additional Code

This section consists of two parts. The first part contains functions that are used multiple times throughout the book (the *Standard Library*), and the second part contains additional code for some chapters to make everything compile.

Standard Library

The compose operator provides function composition.

```
infix operator >>> { associativity left }
func >>> <A, B, C>(f: A -> B, g: B -> C) -> A -> C {
    return { x in g(f(x)) }
}
```

The curry functions turn a function with x arguments into a series of x functions, each accepting one argument.

```
func curry<A, B, C>(f: (A, B) -> C) -> A -> B -> C {
    return { x in { y in f(x, y) } }
}

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) } } }
}
```

The flip function reverses the order of the arguments of the function you pass into it.

```
func flip<A, B, C>(f: (B, A) -> C) -> (A, B) -> C {
    return { (x, y) in f(y, x) }
}
```

The `all` function takes a list, and checks if a given predicate is true for every element.

```
func all<T> (xs : [T], predicate : T -> Bool) -> Bool {
    for x in xs {
        if !predicate(x) {
            return false
        }
    }
    return true
}
```

The `decompose` array method splits the array into an optional head and tail, unless the list is empty.

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

The `iterateWhile` function repeatedly applies a function while 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
}
```

The >>= operator on optionals is the monadic bind, and can be thought of as a `flatMap`.

```
infix operator >>= {}

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

The `Box` class is used to box values and as a workaround to the limitations with generics in the compiler.

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

Chapter 10, Diagrams

The following code is necessary for all the examples in the diagrams chapter to work.

```
extension NSGraphicsContext {
    var cgContext : CGContextRef {
```

```
        let opaqueContext = COpaquePointer(self.graphicsPort)
        return Unmanaged<CGContextRef>.fromOpaque(opaqueContext)
                .takeUnretainedValue()
    }
}

func *(l: CGPoint, r: CGRect) -> CGPoint {
    return CGPointMake(r.origin.x + l.x*r.size.width,
                       r.origin.y + l.y*r.size.height)
}

func *(l: CGFloat, r: CGPoint) -> CGPoint {
    return CGPointMake(l*r.x, l*r.y)
}
func *(l: CGFloat, r: CGSize) -> CGSize {
    return CGSizeMake(l*r.width, l*r.height)
}

func pointWise(f: (CGFloat, CGFloat) -> CGFloat,
               l: CGSize, r: CGSize) -> CGSize {

    return CGSizeMake(f(l.width, r.width), f(l.height, r.height))
}

func pointWise(f: (CGFloat, CGFloat) -> CGFloat,
               l: CGPoint, r:CGPoint) -> CGPoint {

    return CGPointMake(f(l.x, r.x), f(l.y, r.y))
}

func /(l: CGSize, r: CGSize) -> CGSize {
    return pointWise(/, l, r)
}
func *(l: CGSize, r: CGSize) -> CGSize {
```

```
    return pointWise(*, l, r)
}
func +(l: CGSize, r: CGSize) -> CGSize {
    return pointWise(+, l, r)
}
func -(l: CGSize, r: CGSize) -> CGSize {
    return pointWise(-, l, r)
}

func -(l: CGPoint, r: CGPoint) -> CGPoint {
    return pointWise(-, l, r)
}
func +(l: CGPoint, r: CGPoint) -> CGPoint {
    return pointWise(+, l, r)
}
func *(l: CGPoint, r: CGPoint) -> CGPoint {
    return pointWise(*, l, r)
}


extension CGSize {
    var point : CGPoint {
        return CGPointMake(self.width, self.height)
    }
}

func isHorizontalEdge(edge: CGRectEdge) -> Bool {
    switch edge {
        case .MaxXEdge, .MinXEdge:
            return true
        default:
            return false
    }
}
```

```swift
func splitRect(rect: CGRect, sizeRatio: CGSize,
               edge: CGRectEdge) -> (CGRect, CGRect) {

    let ratio = isHorizontalEdge(edge) ? sizeRatio.width
                                       : sizeRatio.height
    let multiplier = isHorizontalEdge(edge) ? rect.width
                                            : rect.height
    let distance : CGFloat = multiplier * ratio
    var mySlice : CGRect = CGRectZero
    var myRemainder : CGRect = CGRectZero
    CGRectDivide(rect, &mySlice, &myRemainder, distance, edge)
    return (mySlice, myRemainder)
}

func splitHorizontal(rect: CGRect,
                     ratio: CGSize) -> (CGRect, CGRect) {

    return splitRect(rect, ratio, CGRectEdge.MinXEdge)
}

func splitVertical(rect: CGRect,
                   ratio: CGSize) -> (CGRect, CGRect) {

    return splitRect(rect, ratio, CGRectEdge.MinYEdge)
}

extension CGRect {
    init(center: CGPoint, size: CGSize) {
        let origin = CGPointMake(center.x - size.width/2,
                                 center.y - size.height/2)
        self.init(origin: origin, size: size)
    }
}
```

```swift
// A 2-D Vector
struct Vector2D {
    let x: CGFloat
    let y: CGFloat

    var point : CGPoint { return CGPointMake(x, y) }

    var size : CGSize { return CGSizeMake(x, y) }
}

func *(m: CGFloat, v: Vector2D) -> Vector2D {
    return Vector2D(x: m * v.x, y: m * v.y)
}

extension Dictionary {
    var keysAndValues: [(Key, Value)] {
        var result: [(Key, Value)] = []
        for item in self {
            result.append(item)
        }
        return result
    }
}

func normalize(input: [CGFloat]) -> [CGFloat] {
    let maxVal = input.reduce(0) { max($0, $1) }
    return input.map { $0 / maxVal }
}
```

Chapter 11, Generators

The following code is necessary for all the examples in the generators chapter to work

```
func map<A, B>(s: SequenceOf<A>, f: A -> B) -> SequenceOf<B> {
    return SequenceOf { map(s.generate(), f) }
}

extension Int: Smaller {
    func smaller() -> GeneratorOf<Int> {
        let result: Int? = self < 0 ? nil : self.predecessor()
        return one(result)
    }
}
```

Chapter 12, Parser Combinators

```
func none<A>() -> SequenceOf<A> {
    return SequenceOf(GeneratorOf { nil } )
}
func one<A>(x: A) -> SequenceOf<A> {
    return SequenceOf(GeneratorOfOne(x))
}

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
    }
}

func flatMap<A, B>(ls: SequenceOf<A>,
                   f: A -> SequenceOf<B>) -> SequenceOf<B> {

    return join(map(ls, f))
}

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

func map<A, B>(var s: SequenceOf<A>, f: A -> B) -> SequenceOf<B> {
    return SequenceOf {  map(s.generate(), f) }
}

func join<A>(s: SequenceOf<SequenceOf<A>>) -> SequenceOf<A> {
    return SequenceOf {
        JoinedGenerator(map(s.generate()) {
            $0.generate()
```

```
        })
    }
}

func +<A>(l: SequenceOf<A>, r: SequenceOf<A>) -> SequenceOf<A> {
    return join(SequenceOf([l, r]))
}

func const<A, B>(x: A) -> B -> A {
    return { _ in x }
}

func prepend<A>(l: A) -> [A] -> [A] {
    return { (x: [A]) in [l] + x }
}

extension String {
    var characters: [Character] {
        var result: [Character] = []
        for c in self {
            result += [c]
        }
        return result
    }
    var slice: Slice<Character> {
        let res = self.characters
        return res[0..<res.count]
    }
}

extension Slice {
    var head: T? {
        return self.isEmpty ? nil : self[0]
    }
```

```
    var tail: Slice<T> {
        if (self.isEmpty) {
            return self
        }
        return self[(self.startIndex+1)..<self.endIndex]
    }

    var decompose: (head: T, tail: Slice<T>)? {
        return self.isEmpty ? nil
                            : (self[self.startIndex], self.tail)
    }
}

extension Character: Printable {
    public var description: String {
       return "\"(self)\""
    }
}

func string(characters: [Character]) -> String {
    var s = ""
    s.extend(characters)
    return s
}

func member(set: NSCharacterSet, character: Character) -> Bool {
    let unichar = (String(character) as NSString).characterAtIndex(0)
    return set.characterIsMember(unichar)
}

func eof<A>() -> Parser<A, ()> {
    return Parser { stream in
        if (stream.isEmpty) {
```

```
            return one(((), stream))
        }
        return none()
    }
}

func testParser<A>(parser: Parser<Character, A>,
                   input: String) -> String {

    var result: [String] = []
    for (x, s) in parser.p(input.slice) {
        result += ["Success, found \(x), remainder: \(Array(s))"]
    }
    return result.isEmpty ? "Parsing failed." : join("\n", result)
}
```

References

Barendregt, H.P. 1984. *The Lambda Calculus, Its Syntax and Semantics*. Studies in Logic and the Foundations of Mathematics. Elsevier.

Bird, Richard. 2010. *Pearls of Functional Algorithm Design*. Cambridge University Press.

Church, Alonzo. 1941. *The Calculi of Lambda-Conversion*. Princeton University Press.

Claessen, Koen, and John Hughes. 2000. “QuickCheck: a Lightweight Tool for Random Testing of Haskell Programs.” In *ACM SIGPLAN Notices*, 268–279. ACM Press. doi:10.1145/357766.351266.

Gibbons, Jeremy, and Oege de Moor, ed. 2003. *The Fun of Programming*. Palgrave Macmillan.

Girard, Jean-Yves. 1972. “Interprétation Fonctionelle Et élimination Des Coupures de L’arithmétique d’ordre Supérieur.” PhD thesis, Université Paris VII.

Harper, Robert. 2012. *Practical Foundations for Programming Languages*. Cambridge University Press.

Hinze, Ralf, and Ross Paterson. 2006. “Finger Trees: a Simple General-Purpose Data Structure.” *Journal of Functional Programming* 16 (02): 197–217. doi:10.1017/S0956796805005769.

Hudak, P., and M.P. Jones. 1994. “Haskell Vs. Ada Vs. C++ Vs. Awk Vs. ... an Experiment in Software Prototyping Productivity.” Research Report YALEU/DCS/RR-1049. New Haven, CT: Department of Computer Science, Yale University.

Hutton, Graham. 2007. *Programming in Haskell*. Cambridge University Press.

McBride, Conor, and Ross Paterson. 2008. "Applicative Programming with Effects." *Journal of Functional Programming* 18 (01): 1–13.

Moggi, Eugenio. 1991. "Notions of Computation and Monads." *Information and Computation* 93 (1): 55–92.

Okasaki, C. 1999. *Purely Functional Data Structures*. Cambridge University Press.

Peyton Jones, Simon. 2001. "Tackling the Awkward Squad: Monadic Input/Output, Concurrency, Exceptions, and Foreign-Language Calls in Haskell." In *Engineering Theories of Software Construction*, edited by Tony Hoare, Manfred Broy, and Ralf Steinbruggen, 180:47. IOS Press.

Pierce, Benjamin C. 2002. *Types and Programming Languages*. MIT press.

Reynolds, John C. 1974. "Towards a Theory of Type Structure." In *Programming Symposium*, edited by B.Robinet, 19:408–425. Lecture Notes in Computer Science. Springer.

———. 1983. "Types, Abstraction and Parametric Polymorphism." *Information Processing*.

Strachey, Christopher. 2000. "Fundamental Concepts in Programming Languages." *Higher-Order and Symbolic Computation* 13 (1-2): 11–49.

Swierstra, S Doaitse. 2009. "Combinator Parsing: a Short Tutorial." In *Language Engineering and Rigorous Software Development*, 252–300. Springer. doi:10.1.1.184.7953.

Wadler, Philip. 1989. "Theorems for Free!" In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 347–359.

———. 1992a. "Comprehending Monads." *Mathematical Structures in Computer Science* 2 (04): 461–493.

———. 1992b. "The Essence of Functional Programming." In *POPL '92: Conference Record of the Nineteenth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1–14. ACM.

Yorgey, Brent. 2009. “The Typeclassopedia.” *The Monad. Reader* 13: 17.

Yorgey, Brent A. 2012. “Monoids: Theme and Variations (Functional Pearl).” In *Proceedings of the 2012 Haskell Symposium*, 105–116. Haskell ’12. Copenhagen, Denmark. doi:10.1145/2364506.2364520.