Functional Programming in C++

Marco Vermeulen
Rúnar Bjarnason
Paul Chiusano

MANNING

MEAP Edition
Manning Early Access Program

Functional Programming with Kotlin

Version 8

Copyright 2020 Manning Publications

For more information on this and other Manning titles go to
manning.com

©Manning Publications Co. To comment go to liveBook

Licensed to John Edwards <john.edwards@jfinancial.co.uk>

welcome

Thanks for joining the early access program of *Functional Programming in Kotlin*. I hope you have lots of fun as you join me in working through this exciting, yet challenging book.

Why is it a challenging book? This book is a port of Manning's Functional Programming in Scala into Kotlin. The original (known as "the Red Book") has a huge following and is revered by many as the authoritative text about functional programming (FP) in the Scala community. In fact, the Red Book is less about Scala than what it is about good FP techniques and principles. It teaches functional programming from first principles, regardless of language.

Why is this book exciting? Kotlin is the new kid on the block, and many see it as their new language of choice on the JVM. Kotlin is gaining in popularity among backend developers, and not just with mobile (Android) developers. Kotlin also has a remarkable number of similarities to Scala; like Scala, it falls into the category of a general-purpose object functional language.

As a result of the increasing popularity of both FP and Kotlin, I decided to bring these two worlds together. Not much has been written about FP in Kotlin so far. This is the perfect opportunity to bring the success of Functional Programming in Scala to all the Kotlin developers who would love to learn functional programming.

The book closely follows the text of Functional Programming in Scala, but all code samples have been adapted and translated to idiomatic Kotlin code. At times we do deviate from the original text due to the languages being different, but the book remains as true to the original text as possible.

The first part of the book slowly eases into the topics by introducing some basic FP concepts, then explores some Kotlin features necessary to writing good FP code. We then explore how to work with data structures, perform error handling, use lazy evaluation, and work with pure functional state.

The second part deals with functional design, where we explore purely functional parallelism, property based testing, and building a parser combinator library.

The third part of the book deals with monoids, monads(!), applicatives, and traversable functors.

The final section deals with advanced topics such as external and local effects, dealing with mutable state, stream processing and incremental IO.

With that said, let's jump right into it! I hope you enjoy this book as much as I am enjoying translating it for you. May it inspire you to write beautiful, pure functional code in every Kotlin project that you work on in the future.

If you have any questions, comments, or suggestions, please share them in Manning's liveBook's Discussion Forum for my book.

—Marco Vermeulen

©Manning Publications Co. To comment go to liveBook

Licensed to John Edwards <john.edwards@jfinancial.co.uk>

brief contents

PART 1: INTRODUCTION TO FUNCTIONAL PROGRAMMING

1 What is functional programming?

2 Getting started with functional programming in Kotlin

3 Functional data structures

4 Handling error without exceptions

5 Strictness and laziness

6 Purely functional state

PART 2: FUNCTIONAL DESIGN AND COMBINATOR LIBRARIES

7 Purely functional parallelism

8 Property-based testing

9 Parser combinators

PART 3: COMMON STRUCTURES IN FUNCTIONAL DESIGN

10 Monoids

11 Monads and functors

12 Applicative and traversable functors

PART 4: EFFECTS AND I/O

13 External effects and I/O

14 Local effects and mutable state

15 Stream processing and incremental I/O

APPENDIXES:

A Exercise hints and tips

B Exercise solutions

C Expansive exercises

D Higher-kinded types

E Type classes

©Manning Publications Co. To comment go to liveBook

Licensed to John Edwards <john.edwards@jfinancial.co.uk>

1 What is functional programming?

This chapter covers:

- Understanding side effects and the problems they pose
- Achieving a functional solution by removing side effects
- Defining what a pure function is
- Proving referential transparency and purity using the substitution model

Most of us started programming using an *imperative style* of coding. What do we mean by this? It means that we give the computer a set of instructions or commands one after the other. As we do so, we are changing the system's state with each step that we take. We are naturally drawn to this approach because of its initial simplicity. On the other hand, as programs grow in size and become more complicated, this seeming simplicity will lead to the very opposite; complexity arises and takes the place of what we initially intended to do. The end result is code that is not maintainable, difficult to test, hard to reason about and (possibly worst of all) full of bugs. The initial velocity that we were able to deliver features in slows down substantially until even a simple enhancement to our program becomes a slow and laborious task.

Functional programming is an alternative style to the *imperative* that addresses the problems mentioned above. In this chapter we will be looking at a simple example where a piece of imperative code with *side effects* (we'll understand what that means shortly) is transformed into the functional style by a sequence of refactoring steps. The eradication of these side effects is one of the core concepts behind functional programming, and so is one of the highlights of this chapter. We will understand the dangers these effects pose and see how to extract them from our code, bringing us back to the safe place of simplicity where we departed from when we initially set out on our journey.

At this point, it's also worth mentioning that this book is about functional programming using

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Kotlin by example to demonstrate the principles of this programming paradigm. Moreover, the focus is not on Kotlin as a language but rather on deriving the concepts used in functional programming. In fact, many of the constructs that we build are not even available in Kotlin but only in third party libraries such as Arrow [1]. This book teaches you functional programming from *first principles* that could be applied to many programming languages, not just to Kotlin.

Something else to keep in mind while reading this book is the *mathematical nature* of the functional programming that we will set out to learn. Many have written about the topic of functional programming, but the kind that we are describing in this book is a bit different. It relies heavily on the *type system* that statically typed languages such as Kotlin provide, and is often called *typed functional programming*. We will also make mention of *category theory*, a branch of mathematics that aligns itself very closely with this style of programming. Due to this mathematical slant, be prepared for words such as *algebra*, *proofs* and *laws*.

Along these lines, this is not a book of recipes or magic incantations. It won't give you quick fixes or fast pragmatic solutions to your everyday problems as a programmer. Instead, it will teach you and equip you with foundational concepts and theory that you can apply to help you arrive at many pragmatic solutions of your own.

Functional programming (FP) is based on a simple premise with far-reaching implications: we construct our programs using only *pure functions*—in other words, functions that have no *side effects*. What are side effects? A function has a side effect if it does something other than simply return a result, for example:

- Modifying a variable beyond the scope of the block where the change occurs
- Modifying a data structure in place
- Setting a field on an object
- Throwing an exception or halting with an error
- Printing to the console or reading user input
- Reading from or writing to a file
- Drawing on the screen

We provide a more precise definition of side effects later in this chapter, but consider what programming would be like without the ability to do these things, or with significant restrictions on when and how these actions can occur. It may be difficult to imagine. How is it even possible to write useful programs at all? If we can't reassign variables, how do we write simple programs like loops? What about working with data that changes, or handling errors without throwing exceptions? How can we write programs that must perform IO, like drawing to the screen or reading from a file?

The answer is that functional programming is a restriction on *how* we write programs, but not on *what* programs we can express. Over the course of this book, we'll learn how to express the core of our programs without side effects, and that includes programs that perform IO, handle errors,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

and modify data. We'll learn how following the discipline of FP is tremendously beneficial because of the increase in *modularity* that we gain from programming with pure functions. Because of their modularity, pure functions are easier to test, reuse, parallelize, generalize, and reason about. Furthermore, pure functions are much less prone to bugs. In this chapter, we look at a simple program with side effects and demonstrate some of the benefits of FP by removing them. We also discuss the benefits of FP more generally and work up to defining two important concepts — *referential transparency* and the *substitution model*.

1.1 The benefits of FP: a simple example

Let's look at an example that demonstrates some of the benefits of programming with pure functions. The point here is just to illustrate some basic ideas that we'll return to throughout this book. Don't worry too much about the Kotlin syntax. As long as you have a basic idea of what the code is doing, that's what's important.

NOTE Since the focus of this book is on FP and not Kotlin, we assume the reader to already have a working knowledge of the language. Consider reading Manning's *Kotlin in Action* for a more comprehensive treatment of the language itself.

1.1.1 A program with side effects

Suppose we're implementing a program to handle purchases at a coffee shop. We'll begin with a Kotlin program that uses side effects in its implementation (also called an impure program).

Listing 1.1 A Kotlin program with side effects

```
class Cafe {

    fun buyCoffee(cc: CreditCard): Coffee {

        val cup = Coffee()   ❶

        cc.charge(cup.price)   ❷

        return cup   ❸
    }
}
```

❶ Instantiate a new cup of `Coffee`

❷ Charge credit card with the coffee's `price`. A side effect!

❸ Return the `Coffee`.

A method call is made on the `charge` method of the credit card resulting in a side effect. Then the `cup` is passed back to the caller of the method.

The line `cc.charge(cup.price)` is an example of a side effect. Charging a credit card involves

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

some interaction with the outside world—suppose it requires contacting the credit card provider via some web service, authorizing the transaction, charging the card, and (if successful) persisting a record of the transaction for later reference. In contrast, our function merely returns a `Coffee` and these other actions are all happening *on the side*, hence the term "side effect." (Again, we define side effects more formally later in this chapter.)

As a result of this side effect, the code is difficult to test. We don't want our tests to actually contact the credit card provider and charge the card! This lack of testability is suggesting a design change: arguably, `CreditCard` shouldn't have any knowledge baked into it about how to contact the credit card provider to actually execute a charge, nor should it have knowledge of how to persist a record of this charge in our internal systems. We can make the code more modular and testable by letting `CreditCard` be agnostic of these concerns and passing a `Payments` object into `buyCoffee`.

Listing 1.2 Adding a payments object

```
class Cafe {
    fun buyCoffee(cc: CreditCard, p: Payments): Coffee {
        val cup = Coffee()
        p.charge(cc, cup.price)
        return cup
    }
}
```

Though side effects still occur when we call `p.charge(cc, cup.price)`, we have at least regained some testability. Payments can be an interface, and we can write a mock implementation of this interface that is suitable for testing. But that isn't ideal either. We're forced to make `Payments` an interface, when a concrete class may have been fine otherwise, and any mock implementation will be awkward to use. For example, it might contain some internal state that we'll have to inspect after the call to `buyCoffee`, and our test will have to make sure this state has been appropriately modified (mutated) by the call to `charge`. We can use a mock framework or similar to handle this detail for us, but this all feels like overkill if we just want to test that `buyCoffee` creates a charge equal to the price of a cup of coffee.

Separate from the concern of testing, there's another problem: it's difficult to reuse `buyCoffee`. Suppose a customer, Alice, would like to order 12 cups of coffee. Ideally we could just reuse `buyCoffee` for this, perhaps calling it 12 times in a loop. But as it is currently implemented, that will involve contacting the payment provider 12 times, authorizing 12 separate charges to Alice's credit card! That adds more processing fees and isn't good for Alice or the coffee shop.

What can we do about this? We could write a whole new function, `buyCoffees`, with special logic for batching up the charges. Here, that might not be such a big deal, since the logic of buyCoffee is so simple, but in other cases the logic we need to duplicate may be non-trivial, and we should mourn the loss of code reuse and composition!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

1.1.2 A functional solution: removing the side effects

The functional solution is to eliminate side effects and have `buyCoffee` return the charge as a value in addition to returning the `Coffee`. The concerns of processing the charge by sending it off to the credit card provider, persisting a record of it, and so on, will be handled elsewhere.

Figure 1.1 A call to buyCoffee, with and without side effect

Here's what a functional solution in Kotlin might look like:

Listing 1.3 A more functional approach to buying coffee

```
class Cafe {
    fun buyCoffee(cc: CreditCard): Pair<Coffee, Charge> {
        val cup = Coffee()
        return Pair(cup, Charge(cc, cup.price))
    }
}
```

Here we've separated the concern of *creating* a charge from the *processing* or *interpretation* of that charge. The `buyCoffee` function now returns a `Charge` as a value along with the `Coffee`. We'll see shortly how this lets us reuse it more easily to purchase multiple coffees with a single transaction. But what is `Charge`? It's a data type we just invented containing a `CreditCard` and an `amount`, equipped with a handy function, `combine`, for combining charges with the same `CreditCard`:

Listing 1.4 `Charge` as a data type

```
data class Charge(val cc: CreditCard, val amount: Float) {   ❶

    fun combine(other: Charge): Charge =   ❷
        if (cc == other.cc)   ❸
            Charge(cc, amount + other.amount)   ❹
        else throw Exception(
            "Cannot combine charges to different cards"
        )
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❶ A data class declaration with constructor and immutable fields.

❷ A `combine` method combining charges for the same credit card.

❸ Ensure it's the same card, otherwise throw an exception.

❹ A new `Charge` is returned, combining the amount of this and the other.

This data type is responsible for holding the values for a `CreditCard` and an amount of `Float`. A handy method is also exposed that allows this `Charge` to be combined with another `Charge` instance. When an attempt is made to combine two charges with a different credit card, an exception will be thrown. The throwing of an exception is not ideal, and we'll discuss more functional ways of handling error conditions in chapter 4.

Now let's look at `buyCoffees` to implement the purchase of `n` cups of coffee. Unlike before, this can now be implemented in terms of `buyCoffee`, as we had hoped.

Listing 1.5 Buying multiple cups with `buyCoffees`

```
class Cafe {

    fun buyCoffee(cc: CreditCard): Pair<Coffee, Charge> = TODO()

    fun buyCoffees(
        cc: CreditCard,
        n: Int
    ): Pair<List<Coffee>, Charge> {

        val purchases: List<Pair<Coffee, Charge>> =
            List(n) { buyCoffee(cc) }   ❶

        val (coffees, charges) = purchases.unzip()   ❷

        return Pair(
            coffees,
            charges.reduce { c1, c2 -> c1.combine(c2) }
        )   ❸
    }
}
```

❶ Create a self-initialized `List`.

❷ Split the list of `Pairs` into two separate lists.

❸ Produce the output pairing `coffees` to a combined single `Charge`.

The example takes two parameters: a `CreditCard`, and the `Int` number of coffees to be purchased. After the `Coffees` have been successfully purchased, they are placed into a `List` data type. The list is initialized using the `List(n) { buyCoffee(cc) }` syntax, where `n` describes the number of coffees, and `{ buyCoffee(cc) }` a function that is used to initialize each element of the list.

An `unzip` is then used to *destructure* the list of pairs into two separate lists, each representing one side of the `Pair`. Destructuring is the process of extracting values from a complex data type. We are now left with the `coffees` list being a `List<Coffee>`, and `charges` being a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

List<Charge>. The final step involves reconstructing the data into the required output. This is done by constructing a Pair of List<Coffee> mapped to the combined Charges for all the Coffees in the list. reduce is an example of a *higher-order function*, which will be properly introduced in chapter 2.

SIDEBAR **Extracting values by Destructuring**

Kotlin allows us to *destructure* objects (also know as *decomposition* or *extraction*). This occurs when values in the *assignment* (the left side) are extracted from the *expression* (the right side). When we want to destructure a Pair into its left and right components, we simply assign the contained values, separated by a comma and surrounded by a pair of braces, (and) :

```
val (left, right) = Pair(1, 2)
assert left == 1
assert right == 2
```

In subsequent code, we can now use these destructured values as we normally use any value in Kotlin. It is also possible to ignore an unwanted destructured value by replacing it with an underscore, _:

```
val (_, right) = Pair(1, 2)
```

Destructuring is not restricted to the Pair type, but can also be used on many others such as List or even data classes.

Overall, this solution is a marked improvement— we're now able to reuse buyCoffee directly to define the buyCoffees function, and both functions are trivially testable without having to define complicated mock implementations of some Payments interface! In fact, the Cafe is now completely ignorant of how the Charge values will be processed. We can still have a Payments class for actually processing charges, of course, but Cafe doesn't need to know about it. Making Charge into a first-class value has other benefits we might not have anticipated: we can more easily assemble business logic for working with these charges. For instance, Alice may bring her laptop to the coffee shop and work there for a few hours, making occasional purchases. It might be nice if the coffee shop could combine these purchases Alice makes into a single charge, again saving on credit card processing fees. Since Charge is first-class, we can now add the following extension method to List<Charge> in order to coalesce any same-card charges:

Listing 1.6 Coalesce the charges

```
fun List<Charge>.coalesce(): List<Charge> =
    this.groupBy { it.cc }.values
        .map { it.reduce { a, b -> a.combine(b) } }
```

All we need to know for now is that we are adding some behavior through the use of an extension method, in this case a coalesce function to List<Charge>. Let's focus on the body

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

of this method. You will notice that we're passing functions as values to the `groupBy`, `map`, and `reduce` functions. If you can't already, you'll learn to read and write one-liners like this over the next several chapters. The statements `{ it.cc }` and `{ a, b -> a.combine(b) }` are syntax for anonymous functions, which we introduce in the next chapter. You may find this kind of code difficult to read because the notation is very compact. But as you work through this book, reading and writing Kotlin code like this will become second nature to you very quickly. This function takes a list of charges, groups them by the credit card used, and then combines them into a single charge per card. It's perfectly reusable and testable without any additional mock objects or interfaces. Imagine trying to implement the same logic with our first implementation of `buyCoffee`!

This is just a taste of why functional programming has the benefits claimed, and this example is intentionally simple. If the series of refactorings used here seems natural, obvious, unremarkable, or standard practice, that's good. FP is merely a discipline that takes what many consider a good idea to its logical endpoint, applying the discipline even in situations where its applicability is less obvious. As you'll learn over the course of this book, the consequences of consistently following the discipline of FP are profound and the benefits enormous. FP is a truly radical shift in how programs are organized at every level— from the simplest of loops to high-level program architecture. The style that emerges is quite different, but it's a beautiful and cohesive approach to programming that we hope you come to appreciate.

SIDEBAR **What about the real world?**

We saw in the case of `buyCoffee` how we could separate the creation of the `Charge` from the interpretation or processing of that `Charge`. In general, we'll learn how this sort of transformation can be applied to *any* function with side effects to push these effects to the outer layers of the program. Functional programmers often speak of implementing programs with a pure core and a thin layer on the outside that handles effects.

But even so, surely at some point we must actually have an effect on the world and submit the `Charge` for processing by some external system. And aren't there other useful programs that necessitate side effects or mutation? How do we write such programs? As we work through this book, we'll discover how many programs that seem to necessitate side effects have some functional analogue. In other cases we'll find ways to structure code so that effects occur but aren't *observable*. (For example, we can mutate data that's declared locally in the body of some function if we ensure that it can't be referenced outside that function, or we can write to a file as long as no enclosing function can observe this occurring.)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

1.2 Exactly what is a (pure) function?

We said earlier that FP means programming with pure functions, and a pure function is one that has no side effects. In our discussion of the coffee shop example, we worked off an informal notion of side effects and purity. Here we'll formalize this notion, to pinpoint more precisely what it means to program functionally. This will also give us additional insight into one of the benefits of functional programming: pure functions are easier to reason about.

A function `f` with input type `A` and output type `B` (written in Kotlin as a single type: `(A) -> B`, pronounced "A to B") is a computation that relates every value `a` of type `A` to exactly one value `b` of type `B` such that `b` is determined solely by the value of `a`. Any changing state of an internal or external process is irrelevant to computing the result `f(a)`. For example, a function `intToString` having type `(Int) -> String` will take every integer to a corresponding string. Furthermore, if it really is a function, it will do nothing else.

Figure 1.2 A pure function, does only what it states without side effect.

In other words, a function has no observable effect on the execution of the program other than to compute a result given its inputs; we say that it has no side effects. We sometimes qualify such functions as *pure functions* to make this more explicit, but this is somewhat redundant. Unless we state otherwise, we'll often use *function* to imply no side effects [2].

You should be familiar with a lot of pure functions already. Consider the addition (+) operator, which resolves to the `plus` function on all integers. It takes an integer value and returns another integer value. For any two given integers, it will always return the same integer value. Another example is the length function of a String in Java, Kotlin, and many other languages where strings can't be modified (are immutable). For any given string, the same length is always returned and nothing else occurs.

We can formalize this idea of pure functions using the concept of referential transparency (RT). This is a property of expressions in general and not just functions. For the purposes of our discussion, consider an expression to be any part of a program that can be evaluated to a result—anything that you could type into the Kotlin interpreter and get an answer. For example,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`2 + 3` is an expression that applies the pure function `plus` on `2` to `3` (which is also an expression). This has no side effect. The evaluation of this expression results in the same value `5` every time. In fact, if we saw `2 + 3` in a program we could simply replace it with the value `5` and it wouldn't change a thing about the meaning of our program.

This is all it means for an expression to be referentially transparent—in any program, the expression can be replaced by its result without changing the meaning of the program. And we say that a function is pure if calling it with RT arguments is also RT. We'll look at some examples next.

SIDEBAR **Referential transparency and purity**

An expression `e` is referentially transparent if, for all programs `p`, all occurrences of `e` in `p` can be replaced by the result of evaluating `e` without affecting the meaning of `p`. A function `f` is pure if the expression `f(x)` is referentially transparent for all referentially transparent `x`.

1.3 Referential transparency, purity, and the substitution model

Let's see how the definition of RT applies to our original `buyCoffee` example:

```
fun buyCoffee(cc: CreditCard): Coffee {
    val cup = Coffee()
    cc.charge(cup.price)
    return cup
}
```

Whatever the return type of `cc.charge(cup.price)`, even if it's `Unit`, is discarded by `buyCoffee`. Thus, the result of evaluating `buyCoffee(aliceCreditCard)` will be merely `cup`, which is equivalent to a new `Coffee()`. For `buyCoffee` to be pure, by our definition of RT, it must be the case that `p(buyCoffee(aliceCreditCard))` behaves the same as `p(Coffee())`, for any `p`. This clearly doesn't hold—the program `Coffee()` doesn't do anything, whereas `buyCoffee(aliceCreditCard)` will contact the credit card provider and authorize a charge. Already we have an observable difference between the two programs.

Referential transparency enforces the rule that everything a function does should be represented by the value that it returns, according to the result type of the function. This constraint enables a simple and natural mode of reasoning about program evaluation called the substitution model. When expressions are referentially transparent, we can imagine that computation proceeds much like we'd solve an algebraic equation. We fully expand every part of an expression, replacing all variables with their referents, and then reduce it to its simplest form. At each step we replace a term with an equivalent one; computation proceeds by substituting equals for equals. In other words, RT enables equational reasoning about programs.

Let's look at two more examples—one where all expressions are RT and can be reasoned about

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

using the substitution model, and one where some expressions violate RT. There's nothing complicated here; we're just formalizing something you likely already understand.

Let's try the following in the Kotlin interpreter (also known as the Read-Eval-Print-Loop or REPL[3]. Note that in Java and in Kotlin, strings are immutable. A "modified" string is really a new string; the old string remains intact:

```
>>> val x = "Hello, World"
res1: kotlin.String = Hello, World

>>> val r1 = x.reversed()   ❶
res2: kotlin.String = dlroW ,olleH

>>> val r2 = x.reversed()   ❶
res3: kotlin.String = dlroW ,olleH
```

❶ `r1` and `r2` evaluate to the same value

Suppose we replace all occurrences of the term `x` with the expression referenced by `x` (its definition), as follows:

```
>>> val r1 = "Hello, World".reversed()   ❶
res4: kotlin.String = dlroW ,olleH

>>> val r2 = "Hello, World".reversed()   ❶
res5: kotlin.String = dlroW ,olleH
```

❶ `r1` and `r2` *still* evaluate to the same value

This transformation doesn't affect the outcome. The values of `r1` and `r2` are the same as before, so `x` was referentially transparent. What's more, `r1` and `r2` are referentially transparent as well, so if they appeared in some other part of a larger program, they could in turn be replaced with their values throughout and it would have no effect on the program.

Now let's look at a function that is *not* referentially transparent. Consider the `append` function on the `java.lang.StringBuilder` class. This function operates on the `StringBuilder` in place. The previous state of the `StringBuilder` is destroyed after a call to append. Let's try this out:

```
>>> val x = StringBuilder("Hello")
res6: kotlin.text.StringBuilder /* = java.lang.StringBuilder */ = Hello

>>> val y = x.append(", World")
res7: java.lang.StringBuilder! = Hello, World

>>> val r1 = y.toString()
res8: kotlin.String = Hello, World

>>> val r2 = y.toString()
res9: kotlin.String = Hello, World
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 1.3 Calling toString() multiple times on a StringBuilder always yields the same result.

So far so good. Now let's see how this side effect breaks RT. Suppose we substitute the call to `append` like we did earlier, replacing all occurrences of `y` with the expression referenced by `y`:

```
>>> val x = StringBuilder("Hello")
res10: kotlin.text.StringBuilder /* = java.lang.StringBuilder */ = Hello

>>> val r1 = x.append(", World").toString()
res11: kotlin.String = Hello, World

>>> val r2 = x.append(", World").toString()
res12: kotlin.String = Hello, World, World
```

Figure 1.4 Calling append() multiple times on a StringBuilder never yields the same result.

This transformation of the program results in a different outcome. We therefore conclude that `StringBuilder.append` is not a pure function. What's going on here is that although `r1` and `r2` look like they're the same expression, they are in fact referencing two different values of the same `StringBuilder`. By the time the second call is made to `x.append`, the first call will already have mutated the object referenced by `x` . If this seems difficult to think about, that's because it is! Side effects make reasoning about program behavior more difficult.

Conversely, the substitution model is simple to reason about since effects of evaluation are purely local (they affect only the expression being evaluated) and we need not mentally simulate sequences of state updates to understand a block of code. Understanding requires only *local reasoning*. We need not mentally track all the state changes that may occur before or after our function's execution to understand what our function will do; we simply look at the function's

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

definition and substitute the arguments into its body. Even if you haven't used the name "substitution model," you have certainly used this mode of reasoning when thinking about your code.

Formalizing the notion of purity this way gives insight into why functional programs are often more modular. Modular programs consist of components that can be understood and reused independently of the whole, such that the meaning of the whole depends only on the meaning of the components and the rules governing their composition; that is, they are composable. A pure function is modular and composable because it separates the logic of the computation itself from "what to do with the result" and "how to obtain the input"; it's a black box. Input is obtained in exactly one way: via the argument(s) to the function. And the output is simply computed and returned. By keeping each of these concerns separate, the logic of the computation is more reusable; we may reuse the logic wherever we want without worrying about whether the side effect being done with the result or the side effect requesting the input are appropriate in all contexts. We saw this in the `buyCoffee` example—by eliminating the side effect of payment processing being done with the output, we were more easily able to reuse the logic of the function, both for purposes of testing and for purposes of further composition (like when we wrote `buyCoffees` and `coalesce`).

1.4 What lies ahead

This short introduction should have given you a good foretaste of what functional programming is. The next chapter will look at the use of higher-order functions, how to write loops in a functional, polymorphic functions, passing anonymous functions and more.

At this point, it is once again worth mentioning that this book is for developers wishing to learn functional programming from first principles. The focus is on those who have a firm grasp of object orientation and imperative programming in a general purpose language, preferably with some prior experience of Kotlin. It should be stressed again that this is not a book *about Kotlin*, but rather about functional programming, *using Kotlin* to illustrate the concepts presented. That said, this book will greatly enhance your ability to apply functional programming techniques and design principals to any Kotlin code that you write in the future.

It is also worth noting that this book is challenging, and will require some effort and diligence to complete on your behalf. A hands-on approach is taken where each chapter has exercises that will help you to understand and internalize the material covered. The exercises build onto each other and it is important that you complete them before moving onto each subsequent section.

If you follow through with this book, you will have a number of new techniques and skills available to use when coding:

- Learn what makes for writing good FP code.
- Work with various data structures.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

- Handle errors in a functional way.
- Utilize lazy evaluation, and work with pure functional state.
- Apply functional design to parallelism, property based testing, and parser combinator libraries.
- Understand and use monoids, monads, applicatives, and traversable functors.
- Confidently work with advanced features such as external and local effects, mutable state, stream processing and incremental IO.

1.5 Summary

- Functional programming results in increased *code modularity*.
- Modularity gained from programming in pure functions results in improved testability, code reuse, parallelization and generalization.
- Modular functional code is easier to reason about.
- Functional programming leads us toward using only *pure* functions.
- A pure function can be defined as a function that has no *side effects*.
- A function has a side effect if it does something *other than returning a result*.
- A function is said to be *referentially transparent* if everything it does is represented by what it returns.
- The *substitution model* can be used to prove referential transparency of a function.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

2 Getting started with functional programming in Kotlin

This chapter covers:

- Defining Higher Order Functions that pass functions as parameters to other functions
- Writing loops in a functional way using recursion
- Abstracting Higher Order Functions to become Polymorphic
- Calling Higher Order Functions with Anonymous Functions
- Following types to implement Polymorphic Functions

In chapter 1, we committed ourselves to using only pure functions. From this commitment, a question naturally arises: how do we write even the simplest of programs? Most of us are used to thinking of programs as sequences of instructions that are executed in order, where each instruction has some kind of effect. In this chapter, we begin learning how to write programs in the Kotlin language just by combining pure functions.

In this chapter we'll introduce some of the basic techniques for how to write functional programs. We'll discuss how to write loops using tail recursive functions, and we'll introduce higher-order functions (HOFs). HOFs are functions that take other functions as arguments and may themselves return functions as their output. We'll also look at some examples of polymorphic HOFs where we use types to guide us toward an implementation.

There's a lot of new material in this chapter. Some of the material related to HOFs may be brain-bending if you have a lot of experience programming in a language without the ability to pass functions around like that. Remember, it's not crucial that you internalize every single concept in this chapter, or solve every exercise. We'll come back to these concepts again from different angles throughout the book, and our goal here is just to give you some initial exposure.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

2.1 Higher-order functions: passing functions to functions

Let's get right into it by covering some of the basics of writing functional programs. The first new idea is this: functions are values. And just like values of other types—such as integers, strings, and lists—functions can be assigned to variables, stored in data structures, and passed as arguments to functions.

When writing purely functional programs, we'll often find it useful to write a function that accepts other functions as arguments. This is called a higher-order function (HOF), and we'll look next at some simple examples to illustrate this. In later chapters, we'll see how useful this capability really is, and how it permeates the functional programming style. But to start, suppose we wanted to adapt our program to print out both the absolute value of a number and the factorial of another number. Here's a sample run of such a program:

```
The absolute value of -42 is 42
The factorial of 7 is 5040
```

2.1.1 A short detour: writing loops functionally

In order to adapt our existing program to demonstrate HOFs, we should introduce some new behaviour. We will do so by adding a new function that calculates the *n*th factorial. To write this simple function, we will need to take a short detour by showing how loops are written in a purely functional way. We do this by introducing *recursion*.

First, let's write `factorial`:

Listing 2.1 A factorial function

```
fun factorial(i: Int): Int {
    fun go(n: Int, acc: Int): Int =   ❶
        if (n <= 0) acc
        else go(n - 1, n * acc)
    return go(i, 1)   ❷
}
```

❶ An *inner* or *local* function definition.

❷ Calling the local function.

NOTE It is common to write functions that are local to the body of another function. In functional programming we shouldn't consider this any stranger than a local integer or string.

The way we write loops functionally, without mutating a loop variable, is with a recursive function. Here we're defining a recursive helper function inside the body of the factorial function. This function will typically handle recursive calls that require an accumulator parameter, or some other signature change that the enclosing function does not have. Such a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

helper function is often called `go` or `loop` by convention. In Kotlin, we can define functions inside any block, including within another function definition. Since it's local, the `go` function can only be referred to from within the scope of the body of the factorial function, just like a local variable would. The definition of `factorial` finally just consists of a call to `go` with the initial conditions for the loop. The arguments to `go` are the state for the loop. In this case, they're the remaining value `n`, and the current accumulated factorial `acc`. To advance to the next iteration, we simply call `go` recursively with the new loop state (here, `go(n-1, n*acc)`), and to exit from the loop, we return a value without a recursive call (here, we return `acc` in the case that `n <= 0`).

Kotlin *does not* manually detect this sort of self-recursion, but requires the function to declare the `tailrec` modifier. This in turn will instruct the compiler to emit the same kind of bytecode as would be found for a while loop[4], provided that the recursive call is in tail position. See the *Tail calls in Kotlin* sidebar for the technical details on this, but the basic idea is that this optimization[5] (or tail call elimination) can be applied when there's no additional work left to do after the recursive call returns.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Tail calls in Kotlin**

A call is said to be in tail position if the caller does nothing other than return the value of the recursive call. For example, the recursive call to `go(n-1,n*acc)` we discussed earlier is in tail position, since the method returns the value of this recursive call directly and does nothing else with it. On the other hand, if we said `1 + go(n-1,n*acc)`, `go` would no longer be in tail position, since the method would still have work to do when `go` returned its result (namely, adding `1` to it).

If all recursive calls made by a function are in tail position, and the function declares the `tailrec` modifier, Kotlin compiles the recursion to iterative loops that don't consume call stack frames for each iteration.

```
fun factorial(i: Int): Int {
    tailrec fun go(n: Int, acc: Int): Int =   ❶
        if (n <= 0) acc
        else go(n - 1, n * acc)   ❷
    return go(i, 1)
}
```

❶ The `tailrec` modifier instructs the compiler to eliminate tail calls.

❷ The function's final declaration is in tail position

If a recursive function has a call in tail position, but does *not* declare itself as `tailrec`, the compiler won't eliminate tail calls, which in turn could result in a `StackOverflowError` being thrown.

In the case where we apply the `tailrec` modifier to a function without its final declaration being in tail position, the compiler will issue a warning:

```
Warning:(19, 9) Kotlin: A function is marked as tail-recursive but no
tail calls are found
```

Even though a warning is better than nothing, a compilation error would have been far more helpful and much safer in this instance.

EXERCISE **2.1**

Write a recursive function to get the *n*th Fibonacci number (en.wikipedia.org/wiki/Fibonacci_number). The first two Fibonacci numbers are 0 and 1. The *n*th number is always the sum of the previous two—the sequence begins 0, 1, 1, 2, 3, 5, 8, 13, 21. Your definition should use a local tail-recursive function.

```
fun fib(i: Int): Int = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

NOTE Kotlin provides us with a convenient way to mark something as *TODO*. A builtin function called `TODO()` is provided that will result in a `NotImplementedError` being thrown when evaluated. The result is that such unimplemented code will always compile, but will result in the exception being thrown as soon as it is evaluated in a program. This gives us a useful way of putting reminder placeholders in our code without affecting the compilation of our code or breaking the build.

2.1.2 Writing our first higher-order function

The code that we have written so far has only one specific purpose. How can we adapt it to handle several scenarios? In this section, we follow an iterative approach where we will crudely introduce a new requirement, then gradually improve the design until we are left with a functional solution using a *higher-order function*.

Figure 2.1 Introducing new behaviour to our program adding functions related to factorials.

Now that we have a function that calculates the *n*th factorial called `factorial`, let's introduce it to the code from before. In addition we'll do some naive duplication by introducing `formatFactorial`, just as we had `formatAbs` for the `abs` function. The new `formatFactorial` function will then in turn be called from `main` as we did for `formatAbs`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 2.2 A simple program including the factorial function

```
object Example {

    private fun abs(n: Int): Int =
        if (n < 0) -n
        else n

    private fun factorial(i: Int): Int {   ❶
        fun go(n: Int, acc: Int): Int =
            if (n <= 0) acc
            else go(n - 1, n * acc)
        return go(i, 1)
    }

    fun formatAbs(x: Int): String {
        val msg = "The absolute value of %d is %d"
        return msg.format(x, abs(x))
    }

    fun formatFactorial(x: Int): String {   ❷
        val msg = "The factorial of %d is %d"
        return msg.format(x, factorial(x))
    }
}

fun main() {
    println(Example.formatAbs(-42))
    println(Example.formatFactorial(7))   ❸
}
```

❶ Add the `factorial` function, making it private.

❷ Add the `formatFactorial` function, public by default.

❸ Call `formatFactorial` from the `main` method.

The two functions, `formatAbs` and `formatFactorial`, are almost identical. If we like, we can generalize these to a single function, `formatResult`, which accepts as an argument the function to apply to its argument:

```
fun formatResult(name: String, n: Int, f: (Int) -> Int): String {
    val msg = "The %s of %d is %d."
    return msg.format(name, n, f(n))
}
```

Our `formatResult` function is a higher-order function (HOF) that takes another function, called `f` (see *Variable-naming conventions* sidebar). We give a type to `f`, as we would for any other parameter. Its type is `(Int) -> Int` (pronounced “int to int” or “int arrow int”), which indicates that `f` expects an integer argument and will also return an integer.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Variable-naming conventions**

It's a common convention to use names like `f`, `g`, and `h` for parameters to a higher-order function. In functional programming, we tend to use very short variable names, even one-letter names. This is usually because HOFs are so general that they have no opinion on what the argument should actually do. All they know about the argument is its type. Many functional programmers feel that short names make code easier to read, since it makes the structure of the code easier to see at a glance.

Our function `abs` from before matches that type; it accepts an `Int` and returns an `Int`. Likewise, `factorial` accepts an `Int` and returns an `Int`, which also matches the `(Int) -> Int` type. We can therefore pass `abs` or `factorial` as the `f` argument to `formatResult` as we do in the following two cases inside our `main` method:

```
fun main() {
    println(formatResult("factorial", 7, ::factorial))
    println(formatResult("absolute value", -42, ::abs))
}
```

A namespace prefix, `::`, is added in order to reference the `factorial` and `abs` functions. You can find more explanation about accessing and namespacing function references in the *Functions as values* sidebar.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Functions as values**

Several ways of passing function parameters exist in Kotlin. Some of these involve passing functions *by reference*, whereas others involve them to be passed *anonymously*. Both of these will seem familiar to anybody who has attempted functional programming in Java 8 or higher.

The first approach involves passing a callable reference to an existing declaration: In this case, we can simply pass through a namespaced reference to a function such as `this::abs` (or simply `::abs`) for a reference in the same object. A fully qualified reference such as `Example::abs` can be used for a function out of scope in a companion object. If we *import* the namespace, we can reference the function directly from out of scope when calling an HOF as we did in the working example:

```
import Example.factorial
...
formatResult("factorial", 7, ::factorial)
```

The second approach might seem equally familiar to someone coming from Java. This involves anonymously instantiating and passing a *function literal* (also called an *anonymous function* or *lambda*) as parameter. Using the `abs` example as before it would look something like this:

```
formatResult("absolute", -42,
    fun(n: Int): Int { return if (n < 0) -n else n }
)
```

This does seem a bit clunky and can be simplified to something more idiomatic like this:

```
formatResult("absolute", -42, { n -> if (n < 0) -n else n })
```

If a lambda function has only *one* parameter, it can even be replaced with the implicit convenience parameter `it`. The final result looks like this:

```
formatResult("absolute", -42, { if (it < 0) -it else it })
```

Even though we have omitted type declarations in the above examples, the types are still vital and are *inferred* in all cases; the lambda must still be of type `(Int) -> Int`, otherwise compilation will fail.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

2.2 Polymorphic functions: abstracting over types

So far we've defined only *monomorphic functions*, or functions that operate on only one type of data. For example, `abs` and `factorial` are specific to arguments of type `Int`, and the higher-order function `formatResult` is also fixed to operate on functions that take arguments of type `Int`. Often, and especially when writing higher-order functions, we want to write code that works for *any* type it's given. These are called *polymorphic functions*. In the chapters ahead, you'll get plenty of experience writing such functions, so here we'll just introduce the idea.

NOTE We're using the term *polymorphism* in a slightly different way than you might be used to if you're familiar with object-oriented programming, where that term usually connotes some form of subtyping or inheritance relationship. There are no interfaces or subtyping here in this example. The kind of polymorphism we're using here is sometimes called *parametric polymorphism*, and is more akin to the generics found in languages like Java.

When applied to functions, we speak of *polymorphic functions* or *generic functions*, although we will be referring to them as the former from here on out.

2.2.1 An example of a polymorphic function

We can often discover polymorphic functions by observing that several monomorphic functions all share a similar structure. For example, the following monomorphic function, `findFirst`, returns the first index in an array where the key occurs, or `-1` if it's not found. It specializes in searching for a `String` in an `Array` of `String` values.

Listing 2.3 Monomorphic function to find a string in an array

```
fun findFirst(ss: Array<String>, key: String): Int {
    tailrec fun loop(n: Int): Int =
        when {
            n >= ss.size -> -1   ❶
            ss[n] == key -> n   ❶
            else -> loop(n + 1)   ❸
        }
    return loop(0)   ❹
}
```

❶ If the end of the loop has been reached without finding the key, return -1.

❷ If the key is found, return its position.

❸ Recursively call the function, incrementing the counter.

❹ Initialise the loop with count `0`.

The details of the code aren't too important here. What's important is that the code for `findFirst` will look almost identical if we're searching for a `String` in an `Array<String>`, an

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`Int` in an `Array<Int>`, or an `A` in an `Array<A>` for any given type `A`. We can write `findFirst` more generally for any type `A` by accepting a function to use for testing a particular `A` value.

Listing 2.4 Polymorphic function to find an element in an array

```
fun <A> findFirst(xs: Array<A>, p: (A) -> Boolean): Int {   ❶
    tailrec fun loop(n: Int): Int =
        when {
            n >= xs.size -> -1
            p(xs[n]) -> n   ❶
            else -> loop(n + 1)
        }
    return loop(0)
}
```

❶ Operate on an array of `A`, take a predicate function operating on individual elements of `A`.

❷ Apply the predicate function to the array element.

Figure 2.2 Transition from a monomorphic to polymorphic function by introducing abstract types

This is an example of a polymorphic function, sometimes called a *generic* function. We're abstracting over the type of the array and the function used for searching it. To write a polymorphic function as a method, we introduce a comma-separated list of type parameters, surrounded by angle brackets (here, just a single `<A>`), following the name of the function, in this case `findFirst`. We can call the type parameters anything we want—`<Foo, Bar, Baz>` and `<TheParameter, another_good_one>` are valid type parameter declarations—though by convention we typically use short, one-letter, uppercase type parameter names like `<A,B,C>`.

The type parameter list introduces *type variables* that can be referenced in the rest of the type signature (exactly analogous to how variables introduced in the parameter list to a function can be referenced in the body of the function). In `findFirst`, the type variable `A` is referenced in two places: the elements of the array are required to have the type `A` (since it's an `Array<A>`), and the `p` function must accept a value of type `A` (since it's a function of type `(A) -> Boolean`). The fact that the same type variable is referenced in both places in the type signature implies that the type must be the same for both arguments, and the compiler will enforce this fact anywhere we try to call `findFirst`. If we try to search for a `String` in an `Array<Int>`, for instance, we'll get a type mismatch error.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 2.2

Implement `isSorted`, which checks whether a singly linked list, `List<A>` is sorted according to a given comparison function. The function is preceded by two *extension properties* that add `head` and `tail` to any `List` value. The `head` property returns the first element of the list, while the `tail` returns all subsequent elements as another `List<A>`. For a refresher on extension properties, refer to the *Extension methods and properties* sidebar.

```
val <T> List<T>.tail: List<T>
    get() = drop(1)

val <T> List<T>.head: T
    get() = first()

fun <A> isSorted(aa: List<A>, order: (A, A) -> Boolean): Boolean = TODO()
```

SIDEBAR Extension methods and properties

Kotlin provides us with a convenient way of adding behaviour (or state) to any type instance. It does so by way of *extension methods and properties*.

We can easily add behaviour to all instances of a given type by adding an extension method as follows:

```
fun Int.show(): String = "The value of this Int is $this"
```

The new `show` method is now available on all instances of `Int`, allowing us make the following call:

```
>>> 1.show()
res1: kotlin.String = The value of this Int is 1
```

Similarly, we can expose properties on all instances:

```
val Int.show: String
    get() = "The value of this Int is $this"
```

As expected, we can access the field as follows:

```
>>> 1.show
res2: kotlin.String = The value of this Int is 1
```

These extension methods and properties are dispatched *statically*, in other words we are not actually modifying the underlying class. An extension function being called is determined by the type of the expression on which the function is invoked, not by the type of the result of evaluating that expression at runtime.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

2.2.2 Calling HOFs with anonymous functions

When using HOFs, it's often convenient to be able to call these functions with *function literals*, rather than having to supply some existing named function. For instance, we can test the `findFirst` function in the REPL as follows:

```
>>> findFirst(arrayOf(7, 9, 13), { i: Int -> i == 9 })
res0: kotlin.Int = 1
```

There is some new syntax here. The expression `arrayOf(7, 9, 13)` is a builtin library function that builds an array. It constructs a new array with three integers in it. We also pass in a function literal as predicate, checking if the implicit integer parameter of this function is equal to 9. The syntax `{ i: Int -> i == 9 }` is a function literal or anonymous function. Instead of defining this function as a method with a name, we can define it inline using this convenient syntax. This particular function takes one argument called `i` of type `Int`, and it returns a `Boolean` indicating whether `x` is equal to `9`. In general, the arguments to the function are declared to the left of the `->` arrow, and we can then use them in the body of the function to the right of the arrow. For example, if we want to write an equality function that takes two integers and checks if they're equal to each other, we could write that like this:

```
>>> { x: Int, y: Int -> x == y }
res1: (kotlin.Int, kotlin.Int) -> kotlin.Boolean = (kotlin.Int, kotlin.Int) -> kotlin.Boolean
```

The `(kotlin.Int, kotlin.Int) -> kotlin.Boolean` notation given by the REPL indicates that the value of `res1` is a function that takes two arguments. When the type of the function's inputs can be inferred by Kotlin from the context, the type annotations on the function's arguments may be elided, for example, `{ x, y -> x < y }`. We'll see an example of this in the next section, and lots more examples throughout this book.

2.3 Following types to implementations

As you might have seen when writing `isSorted`, the universe of possible implementations is significantly reduced when implementing a polymorphic function. If a function is polymorphic in some type `A`, the only operations that can be performed on that `A` are those passed into the function as arguments (or that can be defined in terms of these given operations).[6] In some cases, you'll find that the universe of possibilities for a given polymorphic type is constrained such that only one implementation is possible! Let's look at an example of a function signature that can only be implemented in one way. It's a higher-order function for performing what's called partial application. This function, `partial1`, takes a value and a function of two arguments, and returns a function of one argument as its result. The name comes from the fact that the function is being applied to some but not all of the arguments it requires:

```
fun <A, B, C> partial1(a: A, f: (A, B) -> C): (B) -> C = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

The `partial1` function has three type parameters: `A`, `B`, and `C`. It then takes two arguments. The argument `f` is itself a function that takes two arguments of types `A` and `B` respectively, and returns a value of type `C`. The value returned by `partial1` will also be a function, of type `(B) -> C`. How would we go about implementing this higher-order function? It turns out that there's only one implementation that compiles, and it follows logically from the type signature. It's like a fun little logic puzzle.[7] Let's start by looking at the type of thing that we have to return. The return type of `partial1` is `(B) -> C`, so we know that we have to return a function of that type. We can just begin writing a function literal that takes an argument of type `B`:

```
fun <A, B, C> partial1(a: A, f: (A, B) -> C): (B) -> C =
    { b: B -> TODO() }
```

This can be weird at first if you're not used to writing anonymous functions. Where did that `B` come from? Well, we've just written, "Return a function that takes a value `b` of type `B`." On the right-hand-side of the `->` arrow (where the `TODO()` is now) comes the body of that anonymous function. We're free to refer to the value `b` in there for the same reason that we're allowed to refer to the value `a` in the body of `partial1`.[8] Let's keep going. Now that we've asked for a value of type `B`, what do we want to return from our anonymous function? The type signature says that it has to be a value of type `C`. And there's only one way to get such a value. According to the signature, `C` is the return type of the function `f`. So the only way to get that `C` is to pass an `A` and a `B` to `f`. That's easy:

```
fun <A, B, C> partial1(a: A, f: (A, B) -> C): (B) -> C =
    { b: B -> f(a, b) }
```

And we're done! The result is a higher-order function that takes a function of two arguments and partially applies it. That is, if we have an `A` and a function that needs both `A` and `B` to produce `C`, we can get a function that just needs `B` to produce `C` (since we already have the `A`). It's like saying, "If I can give you a carrot for an apple and a banana, and you already gave me an apple, you just have to give me a banana and I'll give you a carrot." Note that the type annotation on `b` isn't needed here. Since we told Kotlin the return type would be `(B) -> C`, Kotlin knows the type of `b` from the context and we could just write `{ b -> f(a,b) }` as the implementation. Generally speaking, we'll omit the type annotation on a function literal if it can be inferred by Kotlin. The final result being:

```
fun <A, B, C> partial1(a: A, f: (A, B) -> C): (B) -> C =
    { b -> f(a, b) }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 2.3

Let's look at another example, currying,[9] which converts a function `f` of two arguments into a function of one argument that partially applies `f`. Here again there's only one implementation that compiles. Write this implementation.

```
fun <A, B, C> curry(f: (A, B) -> C): (A) -> (B) -> C = TODO()
```

EXERCISE 2.4

Implement `uncurry`, which reverses the transformation of `curry`. Note that since `->` associates to the right, `(A) -> ((B) -> C)` can be written as `(A) -> (B) -> C`.

```
fun <A, B, C> uncurry(f: (A) -> (B) -> C): (A, B) -> C = TODO()
```

Let's look at a final example, *function composition*, which feeds the output of one function to the input of another function. Again, the implementation of this function is fully determined by its type signature.

EXERCISE 2.5

Implement the higher-order function that composes two functions.

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C = TODO()
```

It's all well and good to puzzle together little one-liners like this, but what about programming with a large real-world code base? In functional programming, it turns out to be exactly the same. Higher-order functions like `compose` don't care whether they're operating on huge functions backed by millions of lines of code or functions that are simple one-liners. Polymorphic, higher-order functions often end up being extremely widely applicable, precisely because they say nothing about any particular domain and are simply abstracting over a common pattern that occurs in many contexts. For this reason, programming in the large has much the same flavor as programming in the small. We'll write a lot of widely applicable functions over the course of this book, and the exercises in this chapter are a taste of the style of reasoning you'll employ when writing such functions.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

2.4 Summary

- A *higher-order function* accepts other functions as parameters.
- Loops can be written in a functional way by using *tail call recursion*.
- The compiler can warn us if *tail call elimination* was not successful.
- Generic *polymorphic functions* can be written by introducing *type variables* to functions.
- *Anonymous functions* can be passed as parameters to higher-order functions.
- Types in method signatures can be used to drive implementation of polymorphic functions.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

3 Functional data structures

This chapter covers:

- Defining functional data structures using algebraic data types
- Demonstrating how branching logic can be written in a single expression
- Sharing data using functional data structures
- Using list recursion and generalising to higher-order functions
- Practice writing and generalizing pure functions
- Implementing `List` and `Tree` data structures from first principals

We said in chapter 1 that functional programs don't update variables or modify mutable data structures. The emphasis on keeping variables immutable raises pressing questions: what sort of data structures can we use in functional programming, how do we define them in Kotlin, and how do we operate on them?

In this chapter, we'll learn the concept of functional data structures by writing our own implementations of a *singly linked list* and *tree*. We will also learn about the related processing technique of *matching*, and get lots of practice writing and generalizing pure functions.

This chapter has a lot of exercises, particularly to help with this last point—writing and generalizing pure functions. Some of these exercises may be challenging. Always try your best to solve them by yourself, although helpful tips and pointers may be found at the back of the book in Appendix A. On the occasions when you really get stuck, or want to confirm that your answers are correct, you may consult Appendix B for complete solutions. Try to use this resource only when you absolutely must! All the source code for the samples and exercises are also available in our GitHub repository (github.com/fpinkotlin/fpinkotlin).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

3.1 Defining functional data structures

A functional data structure is operated on using only pure functions. As you may recall from chapter 1, a pure function must not change data in place or perform other side effects. Therefore, functional data structures are by definition immutable. An empty list should be as eternal and immutable as the integer values `3` or `4`. And just as evaluating `3 + 4` results in a new number `7` without modifying either `3` or `4`, concatenating two lists together (the syntax for this is `a + b` for two lists `a` and `b`) yields a new list and leaves the two inputs unmodified.

Doesn't this mean we end up doing a lot of extra copying of the data? Perhaps surprisingly, the answer is no, and we'll talk about exactly why that is later in this section. But first let's examine what's probably the most ubiquitous functional data structure, the *singly linked list*. It serves as a good example due to its simplicity, making it easy to reason about and understand the underlying principles of immutable data structures. Listing 3.1 introduces some new syntax and concepts that we'll talk through in detail.

Listing 3.1 Definition of the singly linked list data structure

```
sealed class List<out A> {   ❶
    // helper functions
}

object Nil : List<Nothing>()   ❷

data class Cons<out A>(
    val head: A,
    val tail: List<A>
) : List<A>()   ❸
```

❶ Sealed definition of data type

❷ The `Nil` implementation of `List`

❸ The `Cons` implementation of `List`

Let's look first at the definition of the data type, which begins with the keywords `sealed class` . Usually, we introduce a data type with the `class` keyword. Here we're declaring a `class`, called `List`, with no instance methods on it. Adding `sealed` in front of the class declaration means that all implementations must be declared in this file. A sealed class is also abstract by default, so can not be instantiated by itself. There are two implementations, or *data constructors*, of `List` declared next, to represent the two possible forms a `List` can take. As the figure shows, a `List` can be empty, denoted by the data constructor `Nil`, or it can be nonempty, denoted by the data constructor `Cons` (traditionally short for construct). A nonempty list consists of an initial element, `head`, followed by a `List` (possibly empty) of remaining elements (the `tail`).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 3.1 The singly linked list, each tail links to the next list element.

Just as functions can be polymorphic, data types can be as well, and by adding the type parameter `<out A>` after `sealed class List` and then using that `A` parameter inside of the `Cons` data constructor, we declare the `List` data type to be polymorphic in the type of elements it contains, which means we can use this same definition for a list of `Int` elements (denoted `List<Int>`), `Double` elements (denoted `List<Double>`), `String` elements (`List<String>`), and so on (the `out` indicates that the type parameter `A` is covariant—see sidebar "More about variance" for more information).

A data constructor declaration gives us a function to construct that form of the data type. Here are a few examples:

```
val ex1: List<Double> = Nil
val ex2: List<Int> = Cons(1, Nil)
val ex3: List<String> = Cons("a", Cons("b", Nil))
```

Using `object Nil`, you can write `Nil` to construct an empty `List`, and the `data class Cons` lets you write `Cons(1, Nil)`, `Cons("a", Cons("b", Nil))` and so on to build singly linked lists of arbitrary lengths . Note that because `List` is parameterized on a type, `A`, these are polymorphic functions that can be instantiated with different types for `A`. Here, `ex2` instantiates the `A` type parameter to `Int`, while `ex3` instantiates it to `String`. The `ex1` example is interesting—`Nil` is being instantiated with type `List<Double>`, which is allowed because the empty list contains no elements and can be considered a list of whatever type we want!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **More about variance**

In the declaration `class List<out A>`, the `out` in front of the type parameter `A` is a variance annotation that signals that `A` is a covariant or "positive" parameter of `List`. This means that, for instance, `List<Dog>` is considered a subtype of `List<Animal>`, assuming `Dog` is a subtype of `Animal`. (More generally, for all types `X` and `Y`, if `X` is a subtype of `Y`, then `List<X>` is a subtype of `List<Y>`). We could leave out the `out` in front of the `A`, which would make `List` invariant in that type parameter.

But notice now that `Nil` extends `List<Nothing>`. `Nothing` is a subtype of all types, which means that in conjunction with the variance annotation, `Nil` can be considered a `List<Int>`, a `List<Double>`, and so on, exactly as we want.

These concerns about variance aren't very important for the present discussion and are more of an artifact of how Kotlin encodes data constructors via subtyping, so don't worry if this is not completely clear right now. It's certainly possible to write code without using variance annotations at all, and function signatures are sometimes simpler (whereas type inference often gets worse). We'll use variance annotations throughout this book where it's convenient to do so, but you should feel free to experiment with both approaches.

If you would like to learn more about generics, including covariance and contravariance in Kotlin, feel free to read the Kotlin documentation at bit.ly/2Op7bmh.

Many other languages provide the feature of *pattern matching* to work with such data types, as in the functions `sum` and `product`. We'll examine how we achieve this with the `when` expression in more detail next.

3.2 Working with functional data structures

Up to this point we have focused our attention on the definition of the most basic functional data structure, the singly linked list. Having this definition isn't of much use unless we actually start *doing* something with it. In this section, you will learn to apply the technique of *matching*, in order to interpret and process the `List` that was defined in section 2.1.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 3.2 Companion object inside sealed definition of `List` data type

```
sealed class List<out A> {   ❶

    companion object {   ❷

        fun <A> of(vararg aa: A): List<A> {   ❸
            val tail = aa.sliceArray(1 until aa.size)
            return if (aa.isEmpty()) Nil else Cons(aa[0], of(*tail))
        }

        fun sum(ints: List<Int>): Int =
            when (ints) {
                is Nil -> 0
                is Cons -> ints.head + sum(ints.tail)
            }

        fun product(doubles: List<Double>): Double =
            when (doubles) {
                is Nil -> 1.0
                is Cons ->
                    if (doubles.head == 0.0) 0.0
                    else doubles.head * product(doubles.tail)
            }

    }
}
```

❶ Definition of `List` data structure

❷ Companion object containing functions

❸ Factory helper function

In order to add some behaviour to the `List` type, a companion object is added to the body of its definition. Any functions defined within the `companion object` block can be called like you would a static method in Java. For instance, the `of` method can be used in the following way to construct a new `List` from the parameters passed in:

```
>>> List.of(1, 2)
res0: chapter3.List<kotlin.Int> = Cons(head=1, tail=Cons(head=2, tail=Nil))
```

This method accepts a parameter qualified by a `vararg` keyword. This means that the function is *variadic* in nature, meaning that we can pass in an arbitrary amount of parameters of the same type in place of that parameter. These values are then bound to the parameter as an array of that type, and can subsequently be accessed in the method body. We don't need to know much more about this, although we explain it in detail in the sidebar.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Variadic functions in Kotlin**

The `of` function in the `List` object is a factory method for creating new `List` instances. This method is also a *variadic function*, meaning it accepts zero or more arguments of type `A`. If no argument is provided, it will result in a `Nil` instance of `List`. If arguments *are* provided, the method will return a `Cons` representing those values.

```
fun <A> of(vararg aa: A): List<A> {
    val tail = aa.sliceArray(1 until aa.size)
    return if (aa.isEmpty()) Nil else Cons(aa[0], List.of(*tail))
}
```

For data types, it is a common idiom to have a variadic `of` method in the companion object to conveniently construct instances of the data type. By calling this function `of` and placing it in the companion object, we can invoke it with syntax like `List.of(1, 2, 3, 4)` or `List.of("hi", "bye")`, with as many values as we want separated by commas.

In the example, the parameter `aa` to the method is marked with a preceding `vararg` keyword, and will subsequently be available as type `Array<out A>` despite having the declared type of `A`. In this case we use the `sliceArray` method of the `Array` type to extract the `tail` as a new `Array`.

It is also possible to pass an array *into* a method as a variadic parameter by using a prefixed *spread operator*, `*`. We have done so with the recursive call to `of` with `*tail` in our example.

Although the details of working with arrays are unimportant in the context of this discussion, more information on this topic can be found in the Kotlin documentation at bit.ly/2VkgpSa.

Let's look in detail at the functions, `sum` and `product`, which we placed in the companion object. Both these definitions make use of a matching technique using the `when` expression:

Listing 3.3 Function definitions in the `List` companion object

```
fun sum(ints: List<Int>): Int =
    when (ints) {
        is Nil -> 0
        is Cons -> ints.head + sum(ints.tail)
    }

fun product(doubles: List<Double>): Double =
    when (doubles) {
        is Nil -> 1.0
        is Cons ->
            if (doubles.head == 0.0) 0.0
            else doubles.head * product(doubles.tail)
    }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

As you might expect, the `sum` function states that the sum of an empty list is `0`, and the sum of a nonempty list is the first element, plus the sum of the remaining elements. Likewise the `product` function states that the product of an empty list is `1.0`, the product of any list starting with `0.0` is `0.0`, and the product of any other nonempty list is the first element multiplied by the product of the remaining elements. Note that these are recursive definitions, which are common when writing functions that operate over recursive data types like `List` (which refers to itself recursively in its `Cons` data constructor).

SIDEBAR **Singletons implemented as companion objects**

The `companion object` block inside a class declares and creates a new Singleton object, which is a `class` with only a single named instance. If you are familiar with Singleton objects in Java, declaring one is a lot more verbose than in Kotlin. A Singleton in Kotlin is also a lot safer than in Java, without the need for double-checked locking to guarantee thread safety within the body of the object. Kotlin has no equivalent to Java's `static` keyword, and a `companion object` is often used in Kotlin where you might use a class with static members in Java.

We'll often declare a companion object nested inside our data type and its data constructors. This results in an object with the same name as the data type (in this case `List`) where we put various convenience methods for creating or working with values of the data type.

If, for instance, we wanted a function `fun <A> fill(n: Int, a: A): List<A>` that created a `List` with `n` copies of the element `a`, the `List` companion object would be a good place to put it. We could have created an object `Foo` if we wanted, but using the `List` companion object makes it clear that all the functions are relevant to working with lists.

In Kotlin, matching is achieved by using the `when` expression, working a bit like a fancy `switch` statement. It matches its argument against all branches sequentially until some branch condition is satisfied. The value of the satisfied branch becomes the value of the overall expression. An `else` branch is evaluated if none of the other branch conditions are satisfied. The `else` branch is mandatory, *unless* the compiler can prove that all possible cases are covered with branch conditions.

Let's look a bit closer at matching. Several variants of this construct may be used to achieve the purpose of matching a value. This includes matching by constant values, expressions, ranges, and types. The `when` construct can even be used as an improved `if-else` expression. We won't be needing all of these variants for our purposes in learning functional programming, so let's only focus on those which will be required.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

3.2.1 The `when` construct for matching by type

For our purposes, the most useful approach to matching is by *type*. The `is` keyword is used to match each logic branch by its concrete type. As an added benefit, the type that is matched on the left is also *smartcast* to the implementation required on the right of the branch. This interesting feature of the `when` construct results in the value being cast to the matched type for further use in the expression side of the branch. Let's explain this by way of example.

Listing 3.4 The `when` construct uses smartcast to cast an abstract type to a concrete implementation

```
val ints = List.of(1, 2, 3, 4)   ❶

fun sum(xs: List<Int>): Int =
    when (xs) {
        is Nil -> 0   ❷
        is Cons -> xs.head + sum(xs.tail)   ❸
    }

fun main() = sum(ints)   ❹
```

❶ A statement declaring an abstract `List`

❷ Match a `Nil` implementation

❸ Smartcast a `Cons` implementation

❹ Invoke `sum` function with list

The value `ints` is of type `List`. In this case it would be a `Cons`, but could have been a `Nil` in the case that an empty `List` was created. When passed into the `when` construct, it assumes the abstract type of `List` until it is matched by one of the logic branches. In this example, a `Nil` match will merely return a `0`, but a `Cons` match will result in some interesting behaviour—when we transition from the left hand side of our branch to the right, the value `ints` is automatically cast to `Cons` so that we can access its members `head` and `tail`! This feature, know as smartcasting, becomes invaluable when working with data types where each sub-type in a class hierarchy may have a distinct constructor containing specific fields.

We must always match by type exhaustively, in our case this would be by all the `sealed` implementations of our base class, `List`. If our match proves *not* to be exhaustive (matching on classes that are not `sealed`, or not listing all the `sealed` variants of the base class), we need to provide the `else` condition as a catch-all expression. In the case of `List`, which is sealed and only has a `Nil` and `Cons` implementation, this is not required.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

3.2.2 The `when` construct as alternative to `if-else` logic

Another good use for the `when` construct is to write simpler `if-else` expressions. When used in this way, no parameter needs to be supplied after the `when` keyword, with each conditional branch acting as a predicate for a matching evaluation. As is the case with `if-else`, the `when` construct is also an expression which can be assigned to a value. As an example, let's look at a simple `if` expression:

Listing 3.5 Logical `if-else` chain used to evaluate expressions

```
val x = Random.nextInt(-10, 10)
val y: String = if (x == 0) {   ❶
    "x is zero"
} else if (x < 0) {   ❶
    "is negative"
} else {   ❸
    "x is positive"
}
```

❶ check if `x` is `0`

❷ check if `x` is negative

❸ otherwise `x` can only be positive

This snippet is simple enough, yet difficult to understand due to all the unnecessary ceremony caused by boilerplate code that surrounds the logic. Using the `when` construct results in something like this:

Listing 3.6 The `when` construct used to evaluate expressions

```
val x = Random.nextInt(-10, 10)
val y: String = when {   ❶
    x == 0 ->   ❶
        "x is zero"
    x < 0 ->   ❷
        "x is negative"
    else ->   ❸
        "x is positive"
}
```

❶ no parameter supplied to `when`

❷ logic branches replacing `if` / `else` statements

❸ catch-all `else` statement

The construct acts upon any variables currently in scope, in this case the random value of `x`. Each logic expression on the left results in a `Boolean` result that leads to the evaluation of one of the branches on the right. Since the entire `when` construct is an expression, the result will be assigned to `y`.

This code is far more elegant and concise, making it easier to read and reason about. The `when` construct is one of the most-used tools in our Kotlin toolbox, and we will continually return to it

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

throughout this book. That said, it does have some drawbacks, lacking some crucial features that other peer languages do support.

3.2.3 Pattern matching and how it differs from Kotlin matching

Matching in Kotlin is not perfect, and falls short of what other languages offer in this space. Languages such as Haskell, Scala and Rust provide us with a feature called *pattern matching*. This is remarkably similar to what we've seen in Kotlin's matching, but has better symantics, more abilities and improved usablity compared to that offered by Kotlin's appraoch. Let's draw a comparison between the matching provided by Kotlin's `when` construct, and the way that these other languages handle this in order to highlight these deficiencies.

The feature of pattern matching gives us the ability to not only *match* on a logic expression, but also to *extract values* from that expression. This extraction, or *destructuring*, plays an important role in functional programming, particularly when working with algebraic data types. To fully understand how pattern matching works, let's take a closer look at how we would write this code in Kotlin using `when`, then how we *wish* we could write it using some Kotlin pseudocode applying this pattern matching technique.

Firstly, let's revisit the `sum` function that we wrote in the companion object of our `List` class:

Listing 3.7 Simple `when` matching in `List` companion object

```
fun sum(xs: List<Int>): Int = when (xs) {
    is Nil -> 0
    is Cons -> xs.head + sum(xs.tail)   ❶
}
```

❶ After matching `Cons`, `xs` is smartcast so `head` and `tail` become visible

The most noticeable problem is that we are accessing the value `xs` inside the evaluation of our branch logic by members as `xs.head` and `xs.tail`. Notice that `xs` is declared as a `List`, which has no `head` or `tail`. The fact that `List` has been smartcast to `Cons` is never explicitly stated, which causes confusion about the ambiguous type of `xs`.

If Kotlin supported pattern matching as provided by other languages, it would allow us to express this as in the following Kotlin pseudocode: .Pattern matching in `List` companion object using pseudocode

```
fun sum(xs: List): Int = when(xs) {
    case Nil -> 0   ❶
    case Cons(head, tail) -> head + sum(tail)   ❷
}
```

❶ First `case` pattern, `Nil` extracts nothing

❷ Second `case` pattern, `Cons(head, tail)` extracts `head` and `tail`

What is most noticeable is a new `case` keyword that is followed by a *pattern* declaration, in this

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

case `Cons(head, tail)`. This pattern is first to be matched, then applied. When the code is executed, each branch pattern will be applied to the object parameter of `when` in sequence. When the branch doesn't match, it is simply passed over. When a match is found, the pattern will be applied, *extracting* any declared fields of that object and making them available on the right hand side of that particular branch.

Consider an object of `List` with the structure `Cons(1, (Cons(2, Nil)))` being passed into our pattern matching construct: Since the first pattern of `Nil` does not match, we fall through to the second pattern. Keep in mind that the `Cons` data class has the following class definition with primary constructor:

```
data class Cons<out A>(val head: A, val tail: List<A>) : List<A>()
```

The constructor `(val head: A, val tail: List<A>)` is now superimposed over the object, and both `head` and `tail` values will be extracted. In this case it would be a `head` of type `Int` with value `1`, and a `tail` of type `List<Int>` with value `Cons(2, Nil)`. These two values are now extracted and made available on the *right hand side* of the condition branch, where they can be used without accessing the original object `xs` that was passed into the `when` construct.

This shift in logic may seem very subtle at first, but has significant impact on how you would approach matching code such as this. It means that we no longer access the matched object `xs` directly, nor do we require any smartcast to occur to access its fields. Instead we interact with its extracted fields directly, not even touching `xs` in our evaluations.

Even though many have asked for the inclusion of pattern matching in the Kotlin language [10], the creators have taken a strong stance against it, claiming that it would make the language too complex. I sincerely hope that it will be included in the language at a future date.

3.3 Data sharing in functional data structures

When data is immutable, how do we write functions that, for example, add or remove elements from a list? The answer is simple. When we add an element `1` to the front of an existing list, say `xs`, we return a new list, in this case `Cons(1,xs)`. Since lists are immutable, we don't need to actually copy `xs`; we can just reuse it. This is called data sharing. Sharing of immutable data often lets us implement functions more efficiently; we can always return immutable data structures without having to worry about subsequent code modifying our data. There's no need to pessimistically make copies to avoid modification or corruption, as such copies would be redundant due to the data structures being immutable.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

NOTE Pessimistic copying can become a problem in large programs. When mutable data is passed through a chain of loosely coupled components, each component has to make its own copy of the data because other components might modify it. Immutable data is always safe to share, so we never have to make copies. We find that in the large, FP can often achieve greater efficiency than approaches that rely on side effects, due to much greater sharing of data and computation.

In the same way, to remove an element from the front of a list `mylist = Cons(x,xs)`, we simply return its `tail`, `xs`. There's no real removing going on. The original list, `mylist`, is still available, unharmed. We say that functional data structures are persistent, meaning that existing references are never changed by operations on the data structure.

Figure 3.2 Data sharing in a singly linked list due to common underlying data structures

Let's try implementing a few different functions for modifying lists in different ways.

The functions we will write in the exercises can be written in two ways, either way is perfectly acceptable. The first approach is to place the functions inside the `List` companion object as we did for `sum` and `product` in the example. In this approach, the method takes the list it is acting upon as its first argument:

```
fun <A> tail(xs: List<A>): List<A> = TODO()

>>> val xs = List.of(1, 2, 3, 4)
>>> List.tail(xs)
```

The other approach involves using extension methods like those introduced in the previous chapter. This will add behaviour to the list type itself, so we can operate on it in the following way:

```
fun <A> List<A>.tail(): List<A> = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
>>> xs.tail()
```

EXERCISE 3.1

Implement the function `tail` for removing the first element of a `List`. Note that the function takes constant time. What are different choices you could make in your implementation if the `List` is `Nil`? We'll return to this question in the next chapter.

```
fun <A> tail(xs: List<A>): List<A> = TODO()
```

EXERCISE 3.2

Using the same idea, implement the function `setHead` for replacing the first element of a `List` with a different value.

```
fun <A> setHead(xs: List<A>, x: A): List<A> = TODO()
```

3.3.1 The efficiency of data sharing

As we have seen in Section 2.3, data sharing often lets us implement operations more efficiently due to the immutability of the underlying data structures that we are dealing with. Let's look at a few examples.

EXERCISE 3.3

Generalize `tail` to the function `drop`, which removes the first `n` elements from a list. Note that this function takes time proportional only to the number of elements being dropped—we don't need to make a copy of the entire `List`.

```
fun <A> drop(l: List<A>, n: Int): List<A> = TODO()
```

EXERCISE 3.4

Implement `dropWhile`, which removes elements from the List prefix as long as they match a predicate.

```
fun <A> dropWhile(l: List<A>, f: (A) -> Boolean): List<A> = TODO()
```

Both `drop` and `dropWhile` employed data sharing to achieve their purposes. A more surprising example of data sharing is this function that adds all the elements of one list to the end of another:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 3.8 The `append` function appends all elements of one list to another

```
fun <A> append(a1: List<A>, a2: List<A>): List<A> =
    when (a1) {
        is Nil -> a2
        is Cons -> Cons(a1.head, append(a1.tail, a2))
    }
```

Note that this definition only copies values until the first list is exhausted, so its runtime and memory usage are determined only by the length of `a1`. The remaining list then just points to `a2`. If we were to implement this same function for two arrays, we'd be forced to copy all the elements in both arrays into the result. In this case, the immutable linked list is much more efficient than an array!

EXERCISE 3.5

Not everything works out so nicely as when we append two lists to each other. Implement a function, `init`, that returns a `List` consisting of all but the last element of a `List`. So, given `List(1, 2, 3, 4)`, `init` will return `List(1, 2, 3)`. Why can't this function be implemented in constant time like `tail`?

```
fun <A> init(l: List<A>): List<A> = TODO()
```

Due to the structure of a singly linked list, any time we want to replace the `tail` of a `Cons`, even if it's the last `Cons` in the list, we must copy all the previous `Cons` objects. Writing purely functional data structures that support different operations efficiently is all about finding clever ways to exploit data sharing. We're not going to cover these data structures here; for now, we're content to use the functional data structures others have written.

3.4 Recursion over lists and generalizing to higher-order functions

Let's look again at the implementations of `sum` and `product`. These two functions seem remarkably similar in what they do and how they go about doing it. Next, we will look at extracting commonalities to derive a higher-order function of these two functions.

To bring the implementations of these two functions closer together, we've simplified the `product` implementation slightly, so as not to include the "short-circuiting" logic of checking for `0.0`:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 3.9 Normalizing `product` by removing short-circuit makes it similar to `sum`

```
fun sum(xs: List<Int>): Int = when (xs) {
    is Nil -> 0
    is Cons -> xs.head + sum(xs.tail)
}

fun product(xs: List<Double>): Double = when (xs) {
    is Nil -> 1.0
    is Cons -> xs.head * product(xs.tail)
}
```

Note how similar these two definitions are. They're operating on different types (`List<Int>` versus `List<Double>`), but aside from this, the only differences are the value to return in the case that the list is empty (`0` in the case of `sum`, `1.0` in the case of `product`), and the operation to combine results (`+` in the case of `sum` and `*` in the case of `product`). Whenever you encounter duplication like this, you can generalize it away by pulling subexpressions out into function arguments. If a subexpression refers to any local variables (the `+` operation summing up two values in `sum`, as well as the `*` operation multiplying two values in `product`), turn the subexpression into a function that accepts these variables as arguments. Let's do that now. Our function will take as arguments the value to return in the case of the empty list, and the function to add an element to the result in the case of a nonempty list.

Listing 3.10 Use `foldRight` as generalization of `product` and `sum`

```
fun <A, B> foldRight(xs: List<A>, z: B, f: (A, B) -> B): B =
    when (xs) {
        is Nil -> z
        is Cons -> f(xs.head, foldRight(xs.tail, z, f))
    }

fun sum2(ints: List<Int>): Int =
    foldRight(ints, 0, { a, b -> a + b })

fun product2(dbs: List<Double>): Double =
    foldRight(dbs, 1.0, { a, b -> a * b })
```

`foldRight` is not specific to any one type of element, and we discover while generalizing that the value that's returned doesn't have to be of the same type as the elements of the list! One way of describing what `foldRight` does is that it replaces the constructors of the list, `Nil` and `Cons`, with `z` and `f`, illustrated here using the substitution model that we learned about in chapter 2:

```
Cons(1, Cons(2, Nil))
f   (1, f   (2, z  ))
```

Let's look at a complete example where we systematically replace evaluations until we arrive at our final result. We'll trace the evaluation of the following declaration using the same technique as we did before:

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))),
    0, { x, y -> x + y })
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

We will repeatedly substitute the definition of `foldRight` for its evaluation. This technique of substitution will be used throughout this book:

```
foldRight(Cons(1, Cons(2, Cons(3, Nil))),
    0, { x, y -> x + y })
1 + foldRight(Cons(2, Cons(3, Nil)), 0,
    { x, y -> x + y })
1 + (2 + foldRight(Cons(3, Nil), 0,
    { x, y -> x + y }))
1 + (2 + (3 + (foldRight(Nil as List<Int>, 0,
    { x, y -> x + y }))))
1 + (2 + (3 + (0)))
6
```

Note that `foldRight` must traverse all the way to the end of the list (pushing frames onto the call stack as it goes) before it can begin collapsing it by applying the anonymous function.

We are using the lambda syntax for passing the anonymous function parameter, `{ x, y -> x + y }` as `f` into each recursive call to `foldRight`. All types for function parameters of `f` can be inferred, so we do not need to provide them for `x` and `y` respectively.

EXERCISE 3.6

Can `product`, implemented using `foldRight`, immediately halt the recursion and return `0.0` if it encounters a `0.0`? Why or why not? Consider how any short-circuiting might work if you call `foldRight` with a large list. This question has deeper implications that we will return to in chapter 5.

EXERCISE 3.7

See what happens when you pass `Nil` and `Cons` themselves to `foldRight`, like this [11]:

```
foldRight(
    List.of(1, 2, 3),
    List.empty<Int>(),
    { x, y -> Cons(x, y) })
```

What do you think this says about the relationship between `foldRight` and the data constructors of `List`?

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

NOTE Simply passing in `Nil` is not sufficient as we are lacking the type information of `A` in this context. As a result we need to express this as `Nil as List<Int>`. Since this is very verbose, a convenience method to circumvent this can be added to the companion object:

```
fun <A> empty(): List<A> = Nil
```

This method will be used in all subsequent listings and exercises to represent an empty `List`.

EXERCISE 3.8

Compute the length of a list using `foldRight`.

```
fun <A> length(xs: List<A>): Int = TODO()
```

WARNING From this point on, the exercises will noticeably increase in difficulty. So much so that in many instances they'll stretch you beyond what you know. If you can't do an exercise, that is perfectly okay and to be expected. Simply try your best to solve each one, and if you *really* don't succeed, refer to Appendix B for the solution, along with an explanation of how to solve the problem where applicable. As stated before, this should only be done as a last resort or when verifying your final solutions. Also, please refrain from skipping any exercises, as each exercise will build upon the knowledge gained by the previous one. The content of the chapter can only be grasped fully by working through each exercise, this being the recurring theme throughout the book.

EXERCISE 3.9

Our implementation of `foldRight` is not tail-recursive and will result in a `StackOverflowError` for large lists (we say it's not stack-safe). Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that is tail-recursive, using the techniques we discussed in the previous chapter. Here is its signature:

```
tailrec fun <A, B> foldLeft(xs: List<A>, z: B, f: (B, A) -> B): B = TODO()
```

EXERCISE 3.10

Write `sum`, `product`, and a function to compute the length of a list using `foldLeft`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 3.11

Write a function that returns the reverse of a list (given `List(1,2,3)` it returns `List(3,2,1)`). See if you can write it using a fold.

EXERCISE 3.12

Can you write `foldLeft` in terms of `foldRight`? How about the other way around? Implementing `foldRight` via `foldLeft` is useful because it lets us implement `foldRight` tail-recursively, which means it works even for large lists without overflowing the stack.

EXERCISE 3.13

Implement `append` in terms of either `foldLeft` or `foldRight`.

EXERCISE 3.14

Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists. Try to use functions we have already defined.

3.4.1 More functions for working with lists

There are many more useful functions for working with lists. We'll cover a few more here, to get additional practice with generalizing functions and to get some basic familiarity with common patterns when processing lists. After finishing this section, you're not going to emerge with an automatic sense of when to use each of these functions. Instead, just get in the habit of looking for possible ways to generalize any explicit recursive functions you write to process lists. If you do this, you'll (re)discover these functions for yourself and develop an instinct for when you'd use each one.

EXERCISE 3.15

Write a function that transforms a list of integers by adding `1` to each element. This should be a pure function that returns a new `List`.

Listing 3.11 3.16

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 3.17

Write a function `map` that generalizes modifying each element in a list while maintaining the structure of the list. Here is its signature: [12]

```
fun <A, B> map(xs: List<A>, f: (A) -> B): List<B> = TODO()
```

EXERCISE 3.18

Write a function `filter` that removes elements from a list unless they satisfy a given predicate. Use it to remove all odd numbers from a `List<Int>`.

```
fun <A> filter(xs: List<A>, f: (A) -> Boolean): List<A> = TODO()
```

EXERCISE 3.19

Write a function `flatMap` that works like `map` except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
fun <A, B> flatMap(xa: List<A>, f: (A) -> List<B>): List<B> = TODO()
```

For instance, `flatMap(List.of(1, 2, 3), { i -> List.of(i, i) })` should result in `List(1, 1, 2, 2, 3, 3)`.

EXERCISE 3.20

Use `flatMap` to implement `filter`.

SIDEBAR Trailing lambda parameters

Kotlin provides some syntactic sugar when passing a lambda parameter into a higher order function. More specifically, if a function takes several parameters, of which the lambda is the *final* parameter, it can be placed outside the parentheses of the parameter list. For instance:

```
flatMap(xs, { x -> List.of(x) } )
```

can be expressed as:

```
flatMap(xs) { x -> List.of(x) }
```

This is know as a *trailing lambda*, and makes for a more fluid and readable expression.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 3.21

Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6)` become `List(5,7,9)`.

EXERCISE 3.22

Generalize the function you just wrote so that it's not specific to integers or addition. Name your generalized function `zipWith`.

3.4.2 Lists in the Kotlin standard library

A `List` implementation already exists in the Kotlin standard library [13]. At this point it's important to note the difference between our `List`, and that provided by the standard library: Kotlin provides a *read-only* `List` instead of one that is truly *immutable* like our implementation. In fact, the underlying implementation of the standard library read-only and mutable lists are one and the same, namely a `java.util.ArrayList`. This pragmatic decision was made for the purpose of Java interoperability.

The only difference between the read-only and mutable variants of the Kotlin list is that the mutable version implements a `MutableList` interface that has methods allowing the adding, updating and deleting of the underlying list elements. `MutableList` extends from `List`, which in turn does not have these mutating methods. The result being a unified implementation with multiple views on the underlying list through the use of interfaces.

Figure 3.3 Kotlin standard library collections inheritance hierarchy showing relationship between mutable and read-only variants

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

There are a number of useful methods on the standard library lists. You may want to try experimenting with these and other methods in the REPL after reading the API documentation. These methods are defined as methods on `List<A>`, rather than as standalone functions as we've done in this chapter:

- `fun take(n: Int): List<A>` —Returns a list consisting of the first `n` elements of `this`
- `fun takeWhile(f: (A) -> Boolean): List<A>` —Returns a list consisting of the longest valid prefix of `this` whose elements all pass the predicate `f`
- `fun all(f: (A) -> Boolean): Boolean` —Returns `true` if and only if all elements of `this` pass the predicate `f`
- `fun any(f: (A) -> Boolean): Boolean` —Returns `true` if any element of this passes the predicate `f`

We recommend that you look through the Kotlin API documentation after finishing this chapter, to see what other functions there are. In particular, look up some of the functions that you've implemented while doing the exercises for this chapter. If you find yourself writing an explicit recursive function for doing some sort of list manipulation, check the `List` API to see if something like the function you need already exists.

3.4.3 Inefficiency on assembling list functions from simpler components

One of the problems with `List` is that, although we can often express operations and algorithms in terms of very general-purpose functions, the resulting implementation isn't always efficient—we may end up making multiple passes over the same input, or else have to write explicit recursive loops to allow early termination.

It is always desirable to implement our functions in the most efficient way possible, doing anything else would be wasteful. Even though we don't have the means of implementing such efficient code yet, we will look at implementing this now with the tools we currently have, then come back to it in chapter 5 where we will work on its efficiency.

EXERCISE 3.23

As an example, implement `hasSubsequence` for checking whether a List contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)`, `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function however comes most naturally. We'll return to this implementation in chapter 5 and hopefully improve on it.

TIP

Any two values `x` and `y` can be compared for equality in Kotlin using the expression `x == y`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE

```
tailrec fun <A> hasSubsequence(xs: List<A>, sub: List<A>): Boolean = TODO()
```

3.5 Trees

The `List` data structure and its implementations we have been dealing with in this chapter are examples of *algebraic data types* (ADTs). An ADT is just a data type defined by one or more data constructors, each of which may contain zero or more arguments[14]. We say that the data type is the sum or union of its data constructors, and each data constructor is the product of its arguments, hence the name algebraic data type. [15]

Just as algebra is fundamental to the whole of mathematics, algebraic data types are fundamental to functional programming languages. They're the primitives upon which all of our richer data structures are built, including the `List` and `Tree` that we derive in this chapter. They can also be seen as the building blocks of functional programming, and give us something to act upon when we execute our programs.

NOTE Somewhat confusingly, ADT is sometimes also used to stand for *abstract data type*. This book will always be referring to *algebraic data type*.

SIDEBAR **ADTs and encapsulation**

One might object that algebraic data types violate encapsulation by making public the internal representation of a type. In FP, we approach concerns about encapsulation differently—we don't typically have delicate mutable state which could lead to bugs or violation of invariants if exposed publicly. Exposing the data constructors of a type is often fine, and the decision to do so is approached much like any other decision about what the public API of a data type should be.

We do typically use ADTs for situations where the set of cases is closed (known to be fixed, denoted by the `sealed` keyword). For `List` and `Tree`, changing the set of data constructors would significantly change what these data types are. `List` is a singly linked list—that is its nature—and the two cases `Nil` and `Cons` form part of its useful public API. We can certainly write code that deals with a more abstract API than `List` (we'll see examples of this later in the book), but this sort of information hiding can be handled as a separate layer rather than being baked into List directly.

You can use algebraic data types to define other data structures. Let's define a simple binary tree data structure:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 3.12 Definition of a binary tree data structure

```
sealed class Tree<out A>

data class Leaf<A>(val value: A) : Tree<A>()

data class Branch<A>(
    val left: Tree<A>,
    val right: Tree<A>
) : Tree<A>()
```

Matching again provides a convenient way of operating over elements of our ADT. Let's try writing a few functions.

EXERCISE 3.24

Write a function `size` that counts the number of nodes (leaves and branches) in a tree.

EXERCISE 3.25

Write a function `maximum` that returns the maximum element in a `Tree<Int>`.

TIP

Kotlin provides a handy builtin function called `maxOf` which determines the maximum of two values. For example, the maximum of `x` and `y` can be determined by `maxOf(x, y)`.

EXERCISE 3.26

Write a function `depth` that returns the maximum path length from the root of a tree to any leaf.

EXERCISE 3.27

Write a function `map`, analogous to the method of the same name on `List`, that modifies each element in a tree with a given function.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 3.28

Generalize `size`, `maximum`, `depth`, and `map` for `Tree`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function. Can you draw an analogy between this `fold` function and the left and right folds for `List`?

```
fun <A, B> fold(ta: Tree<A>, l: (A) -> B, b: (B, B) -> B): B = TODO()

fun <A> sizeF(ta: Tree<A>): Int = TODO()

fun maximumF(ta: Tree<Int>): Int = TODO()

fun <A> depthF(ta: Tree<A>): Int = TODO()

fun <A, B> mapF(ta: Tree<A>, f: (A) -> B): Tree<B> = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Algebraic data types in the standard library**

`Pair` and `Triple` are simple tuple-like classes that can hold two or three typed values consecutively. `Pair`, `Triple` and the data classes are all algebraic data types. Data classes have been covered before, but let's take a closer look at the `Pair` and `Triple` ADTs:

```
>>> val p = Pair("Bob", 42)
>>> p
res0: kotlin.Pair<kotlin.String, kotlin.Int> = (Bob, 42)   ❶

>>> p.first   ❷
res1: kotlin.String = Bob

>>> p.second   ❸
res2: kotlin.Int = 42

>>> val (first, second) = p   ❹
>>> first
res3: kotlin.String = Bob
>>> second
res4: kotlin.Int = 42
```

❶ A pair contains two values of arbitrary type

❷ The first value can be accessed as `first`

❸ The second value can be accessed as `second`

❹ A pair can be destructured

In this example, `Pair("Bob", 42)` is a pair whose type is `Pair<String, Int>`. We can extract the first or second element of this pair using values `first` and `second` on the `Pair` object. It is also possible to destructure a `Pair` into its sum components much like you can do with a data class.

A higher arity[16] variant of the `Pair` is the `Triple`, which works much like we would expect it to with fields `first`, `second` and `third`. These tuple types are a handy device for when the data class with its named fields and multitude of methods seems overkill. Sometimes a simple container of typed values would do just as well, if not better.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

3.6 Summary

- Immutable data structures are objects that can to be acted upon by pure functions.
- A sealed class has a finite amount of implementations, restricting data structure grammar.
- The `when` construct can match typed data structures, and is used for selecting an appropriate outcome evaluation.
- Kotlin matching is useful for working with data structures, but falls short of *pattern matching* supported by other functional languages.
- Data *sharing* through the use of immutable data structures allows safe access without the need for copying structure contents.
- List operations are expressed through recursive, generalized higher-order functions, promoting code re-use and modularity.
- Kotlin standard library Lists are read-only, not immutable, which could lead to data corruption when acted upon by pure functions.
- Algebraic data types (ADTs) are the formal name of immutable data structures, and are modeled by data classes, `Pairs` and `Triples` in Kotlin.
- Both `List` and `Tree` that were developed in this chapter are examples of ADTs.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

4 Handling error without exceptions

This chapter covers:

- Understanding the pitfalls of throwing exceptions, and why they break referential transparency
- Adopting a pure functional approach for handling exceptional cases
- Using the `Option` data type to encode success conditions and ignore failures
- Applying the `Either` data type to encode both success and failure conditions

We noted briefly in chapter 1 that throwing an exception is a side effect, and is undesired behavior. But why do we consider throwing exceptions bad? Why is it not a desired effect? The answer has much to do with a *loss of control*. At the point that an exception is thrown, control is delegated *away* from the program, and the exception is propagated up the call stack. This loss of control means one of two things—the program will be terminated because the exception was not handled, or else some part of the program higher up on the call stack will catch and deal with the exception. The complexity of our program has just escalated dramatically, and in functional programming this loss of control and additional complexity should be avoided at all costs.

If exceptions aren't to be thrown in functional code, how do we deal with exceptional cases instead? The big idea is that we can represent failures and exceptions with ordinary values, and we can write higher-order functions that abstract out common patterns of error handling and recovery. The functional solution, of returning errors as values, is safer and retains referential transparency, and through the use of higher-order functions, we can preserve the primary benefit of exceptions—*consolidation of error-handling logic*. We'll take a closer look at exceptions and discuss some of their problems, after which we will see how to deal with such cases using a functional approach.

For the same reason that we created our own `List` and `Tree` data types in chapter 3, we'll create

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

two Kotlin types, `Option` and `Either` in this chapter. As before, the types that we are creating are not present in the Kotlin standard library, but are freely available in other functional programming languages. They have also been ported from such languages by way of Arrow, a supplementary functional companion library to Kotlin. It is worth taking a look at their documentation, which can be found at arrow-kt.io. The purpose of this chapter is to enhance your understanding of how these types can be used for handling errors. After completing this chapter, you should feel free to use the Arrow version of `Option` and `Either`. Even though the semantics of the Arrow versions might differ somewhat, the idea remains the same.

4.1 The problems with throwing exceptions

Why do exceptions break referential transparency, and why is that a problem? Let's look at a simple example. We'll define a function that throws an exception and then call it.

Listing 4.1 Throwing and catching an exception

```
fun failingFn(i: Int): Int {
    val y: Int = throw Exception("boom")   ❶
    return try {
        val x = 42 + 5
        x + y
    } catch (e: Exception) {
        43   ❷
    }
}
```

❶ Declaration of type `Int` throws `Exception`

❷ Unreachable code, so not returning `43`

Calling `failingFn` from the REPL gives the expected error:

```
>>> chapter4.Listing_4_1.failingFn(12)
java.lang.Exception: boom
    at chapter4.Listing_4_1.failingFn(Listing_4_1.kt:7)
```

We can prove that `y` is not referentially transparent. Recall from Section 1.3 that any RT expression may be substituted with the value it refers to, and this substitution should preserve program meaning. If we substitute `throw Exception("boom!")` for `y` in `x + y`, it produces a different result, because the exception will now be raised inside a `try` block that will catch the exception and return `43`:

```
fun failingFn2(i: Int): Int =
    try {
        val x = 42 + 5
        x + (throw Exception("boom!")) as Int   ❶
    } catch (e: Exception) {
        43   ❷
    }
```

❶ A thrown `Exception` can be annotated with any type, here it is `Int`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❷ Exception is caught, so returning `43`

We can demonstrate this in the REPL:

```
>>> chapter4.Listing_4_1.failingFn2(12)
res0: kotlin.Int = 43
```

Another way of understanding RT is that the meaning of RT expressions *does not depend on context* and may be reasoned about locally, whereas the meaning of non-RT expressions is *context-dependent* and requires more global reasoning. For instance, the meaning of the RT expression `42 + 5` doesn't depend on the larger expression it's embedded in—it's always and forever equal to `47`. But the meaning of the expression `throw Exception("boom!")` is very context-dependent—as we just demonstrated, it takes on different meanings depending on which `try` block (if any) it's nested within.

Exceptions have two main problems:

- As we just discussed, *exceptions break RT and introduce context dependence*, moving us away from the simple reasoning of the substitution model and making it possible to write confusing exception-based code. This is the source of the folklore advice that throwing exceptions should be used only for error handling, not for control flow. In functional programming, we avoid throwing exceptions altogether, except under extreme circumstances where we cannot recover.
- *Exceptions are not type-safe*. The type of `failingFn`, `(Int) -> Int` tells us nothing about the fact that exceptions may occur, and the compiler will certainly not force callers of `failingFn` to make a decision about how to handle those exceptions. If we forget to check for an exception in `failingFn`, a thrown exception won't be detected until runtime.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Higher order functions and the use of checked exceptions**

Java's checked exceptions at least force a decision about whether to handle or re-raise an error, but they result in significant boilerplate for callers. More importantly, *they don't work for higher-order functions*, which can't possibly be aware of the specific exceptions that could be raised by their arguments. For example, consider the `map` function we defined for `List`:

```
fun <A, B> map(xs: List<A>, f: (A) -> B): List<B> =
    foldRightL(
        xs,
        List.empty(),
        { a, xa -> Cons(f(a), xa) })
```

This function is clearly useful, highly generic, and at odds with the use of checked exceptions—we can't have a version of `map` for every single checked exception that could possibly be thrown by `f`. Even if we wanted to do this, how would `map` even know what exceptions were possible? This is why generic code, even in Java, so often resorts to using `RuntimeException` or some common checked `Exception` type.

We'd like an alternative to exceptions without these drawbacks, but we don't want to lose out on the primary benefit of exceptions: they allow us to *consolidate and centralize error-handling logic*, rather than being forced to distribute this logic throughout our codebase. The technique we use is based on an old idea: instead of throwing an exception, we return a value indicating that an exceptional condition has occurred. This idea might be familiar to anyone who has used return codes in C to handle exceptions. But instead of using error codes, we introduce a new generic type for these "possibly defined values" and use higher-order functions to encapsulate common patterns of handling and propagating errors. Unlike C-style error codes, the error-handling strategy we use is *completely type-safe*, and we get full assistance from the type-checker in forcing us to deal with errors, with a minimum of syntactic noise. We'll see how all of this works shortly.

4.2 Problematic alternatives to exceptions

Let's consider a realistic situation where we might use an exception and look at different approaches we could use instead. Here's an implementation of a function that computes the mean of a list, which is undefined if the list is empty:

```
fun mean(xs: List<Double>): Double =
    if (xs.isEmpty())
        throw ArithmeticException("mean of emtpy list!")   ❶
    else xs.sum() / length(xs)   ❷
```

❶ An `ArithmeticException` is thrown on `xs` being empty

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❷ Otherwise return the valid result

The `mean` function is an example of what's called a *partial function*: it's not defined for some inputs. A function is typically partial because it makes some assumptions about its inputs that aren't implied by the input types.[17] You may be used to throwing exceptions in this case, but two other options exist which are *also* not desirable. Let's look at these for our `mean` example before we look at the preferred approach:

4.2.1 Sentinel value

The first possible alternative to throwing an exception is to return some sort of bogus value of type `Double`. We could simply return `xs.sum() / xs.length()` in all cases, and have it return `Double.NaN` when the denominator `xs.length()` is zero. Alternatively, we could return some other sentinel value. In yet other situations, we might return `null` instead of a value of the needed type. This general class of approaches is how error handling is often done in languages without exceptions, and we reject this solution for a few reasons:

- It allows errors to silently propagate—the caller can forget to check this condition and won't be alerted by the compiler, which might result in subsequent code not working properly. Often the error won't be detected until much later in the code.
- It results in a fair amount of boilerplate code at call sites, with explicit `if` statements to check whether the caller has received a "real" result. This boilerplate is magnified if you happen to be calling several functions, each of which uses error codes that must be checked and aggregated in some way.
- It's not applicable to polymorphic code. For some output types, we might not even *have* a sentinel value of that type even if we wanted to! Consider a function like `max`, which finds the maximum value in a sequence according to a custom comparison function: `fun <A> max(xs: List<A>, greater: (A,A) -> Boolean): A`. If the input is empty, we can't invent a value of type `A`. Nor can `null` be used here, since `null` is only valid for non-primitive types, and `A` may in fact be a primitive like `Double` or `Int`.
- It demands a special policy or calling convention of callers—proper use of the `mean` function would require that callers do something other than call `mean` and make use of the result. Giving functions special policies like this makes it difficult to pass them to higher-order functions, which must treat all arguments uniformly.

4.2.2 Supplied default value

The second alternative to throwing an exception is to force the caller to supply an argument that tells us what to do in case we don't know how to handle the input:

```
fun mean(xs: List<Double>, onEmpty: Double) =
    if (xs.isEmpty()) onEmpty   ❶
    else xs.sum() / xs.size()   ❶
```

❶ A default value is provided on `xs` being empty.

❷ Otherwise return the valid result

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

This makes `mean` into a *total function*, taking each value of the input type into exactly one value of the output type. But it still has drawbacks—it requires that *immediate* callers have direct knowledge of how to handle the undefined case and limits them to returning a `Double`. What if `mean` is called as part of a larger computation and we'd like to abort that computation if `mean` is undefined? Or perhaps we'd like to take some completely different branch in the larger computation in this case? Simply passing an `onEmpty` parameter doesn't give us this freedom.

We need a way to defer the decision of how to handle undefined cases so that they can be dealt with at the most appropriate level.

4.3 Encoding success conditions with Option

The preferred approach we alluded to in Section 1.2 is to explicitly represent that a function may not always have an answer in the return type. We can think of this approach as deferring the error-handling strategy to the caller. We will introduce a new type called `Option` to represent such a condition. As we mentioned earlier, this type also exists in other functional languages and libraries, but we'll be re-creating it here for pedagogical purposes:

```
sealed class Option<out A> {
    //helper functions in companion object
}

data class Some<out A>(val get: A) : Option<A>()
object None : Option<Nothing>()
```

`Option` has two cases:

- undefined, in which case it will be `None`
- defined, in which case it will be a `Some`

We can use `Option` for our definition of `mean` like so:

Listing 4.2 Using an `Option` to make the `mean` function pure.

```
fun mean(xs: List<Double>): Option<Double> =
    if (xs.isEmpty()) None   ❶
    else Some(xs.sum() / xs.size())   ❶
```

❶ `None` value is returned on `xs` being empty

❷ `Some` value returned wrapping a valid result

The return type now reflects the possibility that the result may not always be defined. We still always return a result of the declared type (now `Option<Double>`) from our function.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 4.1 Responding to invalid inputs comparing sentinel values to Option

4.3.1 Usage patterns for Option

Partial functions abound in programming, and `Option` (and the `Either` data type that we'll discuss shortly in Section 4.4) is typically how this partiality is dealt with in FP. You won't see `Option` used anywhere in the Kotlin standard library, although you will see its use throughout functional libraries like Arrow. Here are some examples of how they use it:

- Lookup on maps for a given key with `getOption` returns a value wrapped in `Some` if found, else `None` for non-existent values.
- `firstOrNone` and `lastOrNone` defined for lists and other iterables return an `Option` containing the first or last elements of a sequence if it's nonempty.

These aren't the only examples—as functional programmers we'll see `Option` come up in many different situations. What makes `Option` convenient is that we can factor out common patterns of error handling via higher-order functions, freeing us from writing the usual boilerplate that

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

comes with exception-handling code. In this section, we'll cover some of the basic functions for working with `Option`. Our goal is not for you to attain fluency with all these functions, but just to get you familiar enough that you can revisit this chapter and make progress on your own when you have to write some functional code to deal with errors.

SIDEBAR **Nullable types, and how they compare to Option**

Kotlin chose not to introduce the concept of `Option`, with the creators citing instantiation of lightweight wrappers as a performance overhead. The alternative solution for dealing with `null` values was to introduce the concept of the *nullable type*.

The type system differentiates between references that could hold a `null`, and those that can never do so. A parallel type hierarchy was introduced so that *every* type in the type system has an equivalent nullable type. For instance, a value that references a `String` that could potentially be `null` must be of type `String?`. The `?` differentiates it from its non-nullable equivalent `String`. This nullable value needs to be handled at the call site, with the compiler forcing you to deal with the duality of its state.

Handling `null`s at compile time is certainly better than allowing `null` values to propagate and blow up with a `NullPointerException` at runtime, but it still leaves a trail of boilerplate code at every call site where these nullable types need to be handled.

In this book we will be focusing on what we believe to be a more functional approach by using the `Option` data type to represent nullable values. The overhead caused by using such objects is negligible, so we won't be referring to nullable types again from here on out.

BASIC FUNCTIONS ON OPTION

`Option` can be thought of like a `List` that can contain at most one element, and many of the `List` functions we saw earlier have analogous functions on `Option`. Let's look at some of these functions.

We'll do something slightly different than in chapter 3 where we put all our `List` functions in the `List` companion object. Here we'll use extension methods when possible, enhancing the objects so they can be called with the syntax `obj.fn(arg1)` or `obj fn arg1` instead of `fn(obj, arg1)`. This is a stylistic choice with no real significance, and we'll use both styles throughout this book.[18] Let's take a closer look.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 4.3 Enhancing the `Option` data type

```
fun <A, B> Option<A>.map(f: (A) -> B): Option<B> = TODO()  ❶

fun <A, B> Option<A>.flatMap(f: (A) -> Option<B>): Option<B> = TODO()  ❷

fun <A> Option<A>.getOrElse(default: () -> A): A = TODO()  ❸

fun <A> Option<A>.orElse(ob: () -> Option<A>): Option<A> = TODO()  ❹

fun <A> Option<A>.filter(f: (A) -> Boolean): Option<A> = TODO()  ❺
```

❶ Apply `f` to transform value of `A` to `B` if the `Option` is not `None`

❷ Apply `f`, which may fail, to the `Option` if the `Option` is not `None`

❸ Return a default value if the `Option` is `None`

❹ Return a default `Option` if the `Option` is `None`

❺ Convert `Some` to `None` if the predicate `f` is not met.

There is something worth mentioning here. The `default: () -> A` type annotation in `getOrElse` (and the similar annotation in `orElse`) indicates that the argument is a no-args function that returns a type `B`. This is frequently used for implementing lazy evaluation. Don't worry about this for now—we'll talk much more about this concept of *non-strictness* in chapter 5.

EXERCISE 4.1

Implement all of the preceding functions on `Option`. As you implement each function, try to think about what it means and in what situations you'd use it. We'll explore when to use each of these functions next. Here are a few hints for solving this exercise:

- It's fine to use matching, though you should be able to implement all the functions besides `map` and `getOrElse` without resorting to this technique.
- For `map` and `flatMap`, the type signature should be enough to determine the implementation.
- `getOrElse` returns the result inside the `Some` case of the `Option`, or if the `Option` is `None`, returns the given default value.
- `orElse` returns the first `Option` if it's defined; otherwise, it returns the second `Option`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

USAGE SCENARIOS FOR THE BASIC OPTION FUNCTIONS

Although we can explicitly match on an `Option`, we'll almost always use the above higher-order functions. Here, we'll try to give some guidance for when to use each one. Fluency with these functions will come with practice, but the objective here is to get some basic familiarity. Next time you try writing some functional code that uses `Option`, see if you can recognize the patterns these functions encapsulate before you resort to pattern matching.

Let's start with `map`. You can use the `map` function to transform the result inside an `Option`, if it exists. We can think of it as proceeding with a computation on the assumption that an error hasn't occurred; it's also a way of deferring the error handling to later code:

```
data class Employee(
    val name: String,
    val department: String,
    val manager: Option<String>
)

fun lookupByName(name: String): Option<Employee> = TODO()

fun timDepartment(): Option<String> =
    lookupByName("Tim").map { it.department }
```

Figure 4.2 Transform the content of an Option using map

Here, `lookupByName("Tim")` returns an `Option<Employee>`, which we transform using `map` to pull out the `String` representing the department. Note that we don't need to explicitly check the result of `lookupByName("Tim")` we simply continue the computation as if no error occurred, inside the argument to `map`. If `lookupByName("Tim")` returns `None`, this will abort the rest of the computation and `map` will not call the `it.department` function at all.

In our example, the `Employee` has a `manager` of type `Option<String>`. If we want to determine who the manager is using a simple `map` like we did for `department`, it would have left us with an unwieldy `Option<Option<String>>`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
val unwieldy: Option<Option<String>> =
    lookupByName("Tim").map { it.manager }
```

When we apply `flatMap`, it will first map, *then* flatten the result[19] so that we are left with a more useful `Option<String>` representing Tim's manager. That can then in turn be dealt with by `getOrElse` for a more tangible result.

```
val manager: String = lookupByName("Tim")
    .flatMap { it.manager }
    .getOrElse { "Unemployed" }
```

Figure 4.3 Flatten and transform the content of Option<Option>> with flatMap

EXERCISE 4.2

Implement the `variance` function in terms of `flatMap`. If the mean of a sequence is `m`, the variance is the mean of `x` minus `m` to the power of `2` for each element of `x` in the sequence. In code, this will be `(x - m).pow(2)`. The `mean` method developed in Listing 4.2 may be used to implement this. See the definition of variance on Wikipedia (en.wikipedia.org/wiki/Variance#Definition).

```
fun variance(xs: List<Double>): Option<Double> = TODO()
```

As the implementation of `variance` demonstrates, with `flatMap` we can construct a computation with multiple stages, any of which may fail, and the computation will abort as soon as the first failure is encountered, since `None.flatMap(f)` will immediately return `None`, without running `f`.

We can use `filter` to convert successes into failures if the successful values don't match the given predicate. A common pattern is to transform an `Option` via calls to `map`, `flatMap`, and/or `filter`, and then use `getOrElse` to do error handling at the end. This can be demonstrated by

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

continuing with our example from Listing 4.4:

```
val dept: String = lookupByName("Tim")
    .map { it.department }
    .filter { it != "Accounts" }
    .getOrElse { "Unemployed" }
```

`getOrElse` is used here to convert from an `Option<String>` to a `String`, by providing a default department in case the key `"Tim"` didn't exist in the map or if Tim's department was not `"Accounts"`.

`orElse` is similar to `getOrElse`, except that we return another Option if the first is `None`. This is often useful when we need to chain together possibly failing computations, trying the second if the first hasn't succeeded.

A common idiom is to do `o.getOrElse(throw Exception("FAIL"))` to convert the `None` case of an `Option` back to an exception. The general rule of thumb is that we use exceptions only if no reasonable program would ever catch the exception; if for some callers the exception might be a recoverable error, we use `Option` (or `Either`, discussed in Section 4.4) to give them flexibility.

As you can see, returning errors as ordinary values can be convenient and the use of higher-order functions lets us achieve the same sort of consolidation of error-handling logic we would get from using exceptions. Note that we don't have to check for `None` at each stage of the computation—we can apply several transformations and then check for and handle `None` when we're ready. But we also get additional safety, since `Option<A>` is a different type than `A`, and the compiler won't let us forget to explicitly defer or handle the possibility of `None`.

4.3.2 Option composition, lifting, and wrapping exception-oriented APIs

It may be easy to jump to the conclusion that once we start using `Option`, it infects our entire code base. One can imagine how any callers of methods that take or return `Option` will have to be modified to handle either `Some` or `None`. But this doesn't happen, and the reason is that we can *lift* ordinary functions to become functions that operate on `Option`.

For example, the `map` function lets us operate on values of type `Option<A>` using a function of type `(A) -> B`, returning `Option<B>`. Another way of looking at this is that `map` turns a function f of type `(A) -> B` into a function of type `(Option<A>) -> Option<B>`. Let's make this explicit:

Listing 4.4 Lifting a function to work with `Option`s

```
fun <A, B> lift(f: (A) -> B): (Option<A>) -> Option<B> =
    { oa -> oa.map(f) }
```

This tells us that any function that we already have lying around can be transformed (via `lift`) to operate *within* the context of a single `Option` value. Let's look at an example to demonstrate

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

how we can use `lift` on the builtin `kotlin.math.abs` function to get an absolute value of a number:

```
val absO: (Option<Double>) -> Option<Double> =
    lift { kotlin.math.abs(it) }
```

The `kotlin.math` namespace contains various standalone mathematical functions including `abs` , `sqrt`, `exp`, and so on. We didn't need to rewrite the `kotlin.math.abs` function to work with optional values; we just lifted it into the `Option` context after the fact. We can do this for *any* function.

Figure 4.4 Lift a simple function to receive and emit Option types

Let's look at another example. Suppose we're implementing the logic for a car insurance company's website, which contains a page where users can submit a form to request an instant online quote. We'd like to parse the information from this form and ultimately call our rate function:

```
/**
 * Top secret formula for computing an annual car
 * insurance premium from two key factors.
 */
fun insuranceRateQuote(
    age: Int,
    numberOfSpeedingTickets: Int
): Double = TODO()
```

We want to be able to call this function, but if the user is submitting their age and number of speeding tickets in a web form, these fields will arrive as simple strings that we have to (try to) parse into integers. This parsing may fail; given a string, `s`, we can attempt to parse it into an `Int` using `s.toInt()`, which throws a `NumberFormatException` if the string isn't a valid integer:

```
>>> "112".toInt()
res0: kotlin.Int = 112

>>> "hello".toInt()
java.lang.NumberFormatException: For input string: "hello"
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        at java.lang.NumberFormatException.forInputString(NumberFormatException.java:65)
        at java.lang.Integer.parseInt(Integer.java:580)
        at java.lang.Integer.parseInt(Integer.java:615)
```

Let's convert the exception-based API of `toInt()` to `Option` and see if we can implement a function `parseInsuranceRateQuote`, which takes the age and number of speeding tickets as strings, and attempts calling the `insuranceRateQuote` function if parsing both values is successful.

```
fun parseInsuranceQuote(
    age: String,
    speedingTickets: String
): Option<Double> {

    val optAge: Option<Int> = catches { age.toInt() }

    val optTickets: Option<Int> =
        catches { speedingTickets.toInt() }

    //return insuranceRateQuote(optAge, optTickets)   ❶
}

fun <A> catches(a: () -> A): Option<A> =   ❷
    try {
        Some(a())   ❸
    } catch (e: Throwable) {   ❹
        None
    }
```

❶ This does not type-check due to incompatibilities

❷ We accept the `A` argument non-strictly, so we can catch any exceptions that occur while evaluating `a` and convert them to `None`.

❸ Invoke non-strict parameter `a` with `()` inside of `Some`

❹ Note: This discards information about the error `e`. We'll improve on this in Section 4.4 with `Either`.

The `catches` function is a general-purpose function we can use to convert from an exception-based API to an `Option`-oriented API. This uses a non-strict or lazy argument, as indicated by the no-args function definition `() -> A` as type of `a`. We'll discuss laziness in much greater detail in chapter 5, but for now all we need to know is that the lazy parameter can be evaluated by calling `invoke()`, or with the equivalent `()` shorthand notation. In other words, the invocation could have been made as `a.invoke()` or `a()`.

But there's a problem—after we parse `optAge` and `optTickets` into `Option<Int>`, how do we call `insuranceRateQuote`, which currently takes two `Int` values? Do we have to rewrite `insuranceRateQuote` to take `Option<Int>` values instead? No, and changing `insuranceRateQuote` would be entangling concerns, forcing it to be aware that a prior computation may have failed, not to mention that we may not have the ability to modify `insuranceRateQuote`—perhaps it's defined in a separate module that we don't have access to.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Instead, we'd like to lift `insuranceRateQuote` to operate in the context of two optional values. We could do this using explicit pattern matching in the body of `parseInsuranceRateQuote`, but that's going to be tedious.

EXERCISE 4.3

Write a generic function, `map2` that combines two `Option` values using a binary function. If either `Option` value is `None`, then the return value is too. Here is its signature:

```
fun <A, B, C> map2(a: Option<A>, b: Option<B>, f: (A, B) -> C): Option<C> =
    TODO()
```

With `map2`, we can now implement `parseInsuranceRateQuote` as follows:

```
fun parseInsuranceQuote(
    age: String,
    speedingTickets: String
): Option<Double> {

    val optAge: Option<Int> = catches { age.toInt() }

    val optTickets: Option<Int> =
        catches { speedingTickets.toInt() }

    return map2(optAge, optTickets) { a, t ->
        insuranceRateQuote(a, t)
    }
}

fun <A> catches(a: () -> A): Option<A> =   ❶
    try {
        Some(a())   ❷
    } catch (e: Throwable) {   ❸
        None
    }
```

Using `map2` now allows the existing `insuranceRateQuote` function to be used. Nonetheless, one drawback still prevails in that if any or both of the `Option`s are `None`, an overall `None` will be returned, and so have lost the knowledge of which has failed.

Nevertheless, the `map2` function means that we never need to modify any existing functions of two arguments to make them "`Option`-aware." We can lift them to operate in the context of `Option` after the fact. Can you already see how you might define `map3`, `map4`, and `map5`? Let's look at a few other similar cases.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 4.4

Write a function, `sequence` that combines a list of `Option`s into one `Option` containing a list of all the `Some` values in the original list. If the original list contains `None` even once, the result of the function should be `None`; otherwise the result should be `Some` with a list of all the values. Here is its signature:[20]

```
fun <A> sequence(xs: List<Option<A>>): Option<List<A>> = TODO()
```

Sometimes we'll want to map over a list using a function that might fail, returning `None` if applying it to any element of the list returns `None`. For example, what if we have a whole list of `String` values that we wish to parse to `Option<Int>`? In that case, we can simply sequence the results of the `map`:

```
fun parseInts(xs: List<String>): Option<List<Int>> =
    sequence(xs.map { str -> catches { str.toInt() } })
```

Unfortunately, this is inefficient, since it traverses the list twice, first to convert each `String` to an `Option<Int>`, and a second pass to combine these `Option<Int>` values into an `Option<List<Int>>`. Wanting to sequence the results of a `map` this way is a common enough occurrence to warrant a new generic function, `traverse`.

EXERCISE 4.5

Implement the `traverse` function. It's fairly straightforward to do using `map` and `sequence`, but try for a more efficient implementation that only looks at the list once. When complete, implement `sequence` by using `traverse`.

```
fun <A, B> traverse(
    xa: List<A>,
    f: (A) -> Option<B>
): Option<List<B>> = TODO()
```

After seeing so many examples, we can conclude that you should *never* have to modify any existing functions to work with optional values. Between `map`, `lift`, `sequence`, `traverse`, `map2`, `map3`, and so on, you have *all* the tools available to you to deal with such cases.

4.3.3 For comprehensions with Option using Arrow fx

Many languages have a feature called *for-comprehensions*, or *monad-comprehensions*. This concept can be described as a construct that applies syntactic sugar over a series of `flatMap` and `map` calls, yielding a final result. The for-comprehension has a far more pleasing and concise syntax that resembles imperative code, as apposed to dealing with a sequence of nested calls.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Kotlin does not provide us with a for-comprehension out of the box, but fortunately Arrow has this covered. In Arrow it is known as an *fx*, but it works in a very similar fashion to what other languages provide. Let's take a look at how we can implement `map2` using the `fx` method on `Option`. Consider the following code as implementation to `map2`:

```
fun <A, B, C> map2(
    oa: Option<A>,
    ob: Option<B>,
    f: (A, B) -> C
): Option<C> =
    oa.flatMap { a ->
        ob.map { b ->
            f(a, b)
        }
    }
```

Using the `fx` function provided by Arrow, this can now be distilled into something far more expressive and seemingly imperative:

```
fun <A, B, C> map2(
    oa: Option<A>,
    ob: Option<B>,
    f: (A, B) -> C
): Option<C> =
    Option.fx {
        val (a) = oa
        val (b) = ob
        f(a, b)
    }
```

A for-comprehension consists of a sequence of statements, like `val (a) = oa` and `val (b) = ob`, followed by an expression like `f(a, b)` that yields the result. The compiler desugars these statements into `flatMap` calls, with the final expression being converted to a call to `map`.

It is worth noting that this style of for-comprehension will not work with our `Option` type, as the supported classes need to be instances of Arrow's `Monad` type class. We will learn about Monads and type classes later in the book, but for now suffice to say that this will only work with Arrow's supported types such as `Option`, `Either`, `List`, `State` and `IO` to mention but a few.

As you become more comfortable using `flatMap` and `map`, you should feel free to begin using for-comprehensions in place of explicit calls to these combinators.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

4.4 Encoding success and failure conditions with Either

As we alluded earlier, the big idea in this chapter is that we can represent failures and exceptions with ordinary values, and write functions that abstract out common patterns of error handling and recovery. `Option` isn't the only data type we could use for this purpose, and although it gets used frequently, it's rather simplistic. One thing you may have noticed with `Option` is that it doesn't tell us anything about what went wrong in the case of an exceptional condition. All it can do is give us `None`, indicating that there's no value to be had. But sometimes we want to know more. For example, we might want a `String` that gives more information, or if an exception was raised, we might want to know what that error actually was.

We can craft a data type that encodes whatever information we want about failures. Sometimes just knowing whether a failure occurred is sufficient, in which case we can use `Option`; other times we want more information. In this section, we'll walk through a simple extension to `Option`, the `Either` data type, which lets us track a *reason* for the failure. Let's look at its definition:

Listing 4.5 The `Either` data type

```
sealed class Either<out E, out A>

data class Left<out E>(val value: E) : Either<E, Nothing>()
data class Right<out A>(val value: A) : Either<Nothing, A>()
```

`Either` has only two cases, just like `Option`. The essential difference is that both cases carry a value. The `Either` data type represents, in a very general way, values that can be one of two things. We can say that it's a *disjoint union* of two types. When we use it to indicate success or failure, by convention the `Right` constructor is reserved for the success case (a pun on "right," meaning correct), and `Left` is used for failure. We've given the left type parameter the suggestive name `E` (for `error`).[21]

Let's look at the `mean` example again, this time returning a `String` in case of failure:

```
fun mean(xs: List<Double>): Either<String, Double> =
    if (xs.isEmpty())
        Left("mean of empty list!")
    else Right(xs.sum() / xs.size())
```

Sometimes we might want to include more information about the error, for example a stack trace showing the location of the error in the source code. In such cases we can simply return the exception in the `Left` side of an `Either`:

```
fun safeDiv(x: Int, y: Int): Either<Exception, Int> =
    try {
        Right(x / y)
    } catch (e: Exception) {
        Left(e)
    }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

To help create a new `Either` we will once again write a function called `catches` which factors out this common pattern of converting thrown exceptions to values.

Listing 4.6 A `catches` function converting exceptions to `Either`

```
fun <A> catches(a: () -> A): Either<Exception, A> =
    try {
        Right(a())
    } catch (e: Exception) {
        Left(e)
    }
```

EXERCISE 4.6

Implement versions of `map`, `flatMap`, `orElse`, and `map2` on `Either` that operate on the `Right` value.

```
fun <E, A, B> Either<E, A>.map(f: (A) -> B): Either<E, B> = TODO()

fun <E, A, B> Either<E, A>.flatMap(f: (A) -> Either<E, B>): Either<E, B> =
    TODO()

fun <E, A> Either<E, A>.orElse(
    f: () -> Either<E, A>
): Either<E, A> = TODO()

fun <E, A, B, C> map2(
    ae: Either<E, A>,
    be: Either<E, B>,
    f: (A, B) -> C
): Either<E, C> = TODO()
```

4.4.1 For comprehensions with Either using Arrow fx

In this section, we will focus on writing elegant for-comprehensions with `Either`. As was the case with `Option`, a variant of `Either` also exists in Arrow that has the ability to be used in such for-comprehensions. Let's look at the following extensive example to demonstrate how this works.

Listing 4.7 Using `Either` in for-comprehensions

```
suspend fun String.parseToInt(): Either<Throwable, Int> =   ❹
    Either.catch { this.toInt() }   ❷

suspend fun parseInsuranceRateQuote(   ❸
    age: String,
    numberOfSpeedingTickets: String
): Either<Throwable, Double> {
    val ae = age.parseToInt()   ❹
    val te = numberOfSpeedingTickets.parseToInt()
    return Either.fx {   ❺
        val (a) = ae   ❻
        val (t) = te
        insuranceRateQuote(a, t)   ❼
    }
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❶ Add `parseToInt` extension method to `String`

❷ Use the `Either.catch` method to produce an `Either<Throwable, Int>`

❸ Method marked `suspended`, meaning its child process could block

❹ Use extension method to produce an `Either`

❺ Open for-comprehension with `Either.fx`

❻ `flatMap` right-biased `Either` by destructuring

❼ Return final evaluation of `insuranceRateQuote` as `Either.Right` on success

This example has a lot going on, so let's work through it slowly. Firstly, we declare an extension function that adds a `parseInt` method to all `String` instances. This function will handle any exceptions thrown by the `toInt` method on `String`, and will automatically return an `Either.Left` containing the thrown exception, or else an `Either.Right` containing the successfully parsed value. Because this could potentially be a blocking operation, we need to mark the function with a `suspend` keyword. A *suspending function* is just a regular Kotlin function with an additional `suspend` modifier which indicates that the function can be suspended on the execution of a long running child process.

The `parseInsuranceQuote` method will in turn use this extension function to parse both its `String` parameters into `Either<Throwable, Int>`. Both of these parameters could result in a failure that results in an `Either.Left` containing the exception, and are subsequently `flatMap` ped from within the for-comprehension marked by the `fx` block.

The final call to `insuranceRateQuote` will only ever be called if both instances of `Either` were `Right`. This would result in an `Either.Right<Double>`. On the other hand, if either or both of these instances were an `Either.Left<Throwable>`, the *first* of these would be returned to the caller of the method thus short-circuiting the rest of the for-comprehension. The right side of the `Either` always takes precedence so it is said to be *right-biased*.

Now we get information about the actual exception that occurred, rather than just getting back `None` in the event of a failure.

EXERCISE 4.7

Implement `sequence` and `traverse` for `Either`. These should return the first error that's encountered, if there is one.

As a final example, here's an application of `map2`, where the function `mkPerson` validates both the given name and the given age before constructing a valid `Person`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 4.8 Using `Either` to validate data

```
data class Person(val name: Name, val age: Age)

data class Name(val value: String)
data class Age(val value: Int)

fun mkName(name: String): Either<String, Name> =
    if (name.isBlank()) Left("Name is empty.")
    else Right(Name(name))

fun mkAge(age: Int): Either<String, Age> =
    if (age < 0) Left("Age is out of range.")
    else Right(Age(age))

fun mkPerson(name: String, age: Int): Either<String, Person> =
    map2(mkName(name), mkAge(age)) { n, a -> Person(n, a) }
```

EXERCISE 4.8

In the implementation found in Listing 4.8, `map2` is only able to report one error, even if both the name and the age are invalid. What would you need to change in order to report *both* errors? Would you change `map2` or the signature of `mkPerson`? Or could you create a new data type that captures this requirement better than `Either` does, with some additional structure? How would `orElse`, `traverse`, and `sequence` behave differently for that data type?

4.5 Summary

- Throwing exceptions is not desirable and breaks referential transparency.
- The throwing of exceptions should be reserved only for extreme situations where recovery is not possible.
- You can achieve purely functional error handling by using data types that encapsulate exceptional cases.
- The `Option` data type is convenient for encoding a simple success condition as `Some` or a failure as an empty `None`.
- The `Either` data type can encode a success condition as `Right` *or* a failure condition as `Left`.
- Functions that are prone to throwing exceptions may be *lifted* to be compliant with `Option` and `Either` types.
- A series of `Option` or `Either` operations may be halted on the first failure encountered.
- The *for-comprehension* is a construct that allows fluid expression of a series of combinator calls.
- The Arrow library, a functional companion to Kotlin, has both `Option` and `Either` constructs that allow the use of for-comprehensions through *binding methods* to simplify code.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

5 Strictness and laziness

This chapter covers:

- Understanding the difference between strict and non-strict functions
- Implementing a lazy list data type
- Memoizing streams to avoid recomputation
- Inspecting streams for the purpose of visualizing and testing
- Separating program description from evaluation
- Consuming infinite streams and grasping corecursion

Kotlin, like most modern programming languages uses *strict* evaluation by default. That is, it allows only functions whose parameters must be evaluated completely before they may be called. In all our examples so far, we have focused on this evaluation strategy, also know as *eager* or *greedy* evaluation. In fact, this is what we have been using while deriving data types such as `List`, `Option` and `Either`. We will look at a more formal definition of strictness later, but what does strict evaluation really imply in the real world?

Strictly evaluated expressions are evaluated at the moment that they are bound to a variable. This includes when they are passed to a function as parameter. This strategy is acceptable if we are merely assigning a simple value, but what if our expression performs some expensive or complex computation to determine its value? And taking this a step further, what if this expensive computation is to be used in expressing *all* elements of a list data type, where we might only need the first few elements?

The notion of *non-strict* or *lazy* evaluation comes to the rescue here—the value is only computed at the point where it is actually referenced, not where it is declared. We are no longer greedy in performing all the calculations, but instead compute them on demand.

In this chapter, we will be looking more closely at the concept of non-strict evaluation, and what

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

the implications of applying this strategy are. We will also be building an algebraic data type that models this concept closely, allowing us to perform all the same operations as what we did on the former data types that we implemented. We will see how these operations help us to bring about a separation of concerns between declaration of a computation and its eventual evaluation. Lastly, we will delve into how to produce infinite data streams using *corecursion*, a technique of lazily generating infinite sequences of values based on what the streams themselves produce.

In chapter 3 we talked about purely functional data structures, using singly linked lists as an example. We covered a number of bulk operations on lists—`map`, `filter`, `foldLeft`, `foldRight`, `zipWith`, and so on. We noted that each of these operations makes its own pass over the input and constructs a fresh list for the output.

Imagine if you had a deck of cards and you were asked to remove the odd-numbered cards and then flip over all the queens. Ideally, you'd make a single pass through the deck, looking for queens and odd-numbered cards at the same time. This is more efficient than removing the odd cards and then looking for queens in the remainder. And yet the latter is what Kotlin is doing in the following snippet of code:

```
>>> List.of(1, 2, 3, 4).map { it + 10 }.filter { it % 2 == 0 }.map { it * 3 }
res0: kotlin.collections.List<kotlin.Int> = [36, 42]
```

In this expression, `map { it + 10 }` will produce an intermediate list that then gets passed to `filter { it % 2 == 0 }`, which in turn constructs a list that gets passed to `map { it * 3 }`, which then produces the final list. In other words, each transformation will produce a temporary list that only ever gets used as input to the next transformation and is then immediately discarded.

Think about how this program will be evaluated. If we manually produce a trace of its evaluation, the steps would look like something as follows.

Listing 5.1 Evaluation trace of operations on a strict list implementation

```
List.of(1, 2, 3, 4)
    .map { it + 10 }.filter { it % 2 == 0 }.map { it * 3 }
List.of(11, 12, 13, 14)
    .filter { it % 2 == 0 }.map { it * 3 }
List.of(12, 14)
    .map { it * 3 }
List.of(36, 42)
```

Here we're showing the result of each substitution performed to evaluate our expression. For example, to go from the first line to the second, we've replaced `List.of(1,2,3,4).map { it + 10 }` with `List.of(11,12,13,14)`, based on the definition of `map`.[22] This view makes it clear how the calls to `map` and `filter` each perform their own traversal of the input and allocate lists for the output. Wouldn't it be nice if we could somehow fuse sequences of transformations like this into a single pass and avoid creating temporary data structures? We could rewrite the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

code into a `while` loop by hand, but ideally we'd like to have this done automatically while retaining the same high-level compositional style. We want to compose our programs using higher-order functions like `map` and `filter` instead of writing monolithic loops.

It turns out that we can accomplish this kind of automatic loop fusion through the use of *non-strictness* (or, less formally, *laziness*). In this chapter, we'll explain exactly what this means, and we'll work through the implementation of a lazy list type that fuses sequences of transformations. Although building a "better" list is the motivation for this chapter, we'll see that non-strictness is a fundamental technique for improving on the efficiency and modularity of functional programs in general.

5.1 Strict and non-strict functions

Before we get to our example of lazy lists, we need to cover some basics. What are strictness and non-strictness, and how can we express these concepts in Kotlin?

Non-strictness is a property of a function. To say a function is non-strict just means that the function may choose not to evaluate one or more of its arguments. In contrast, a *strict* function always evaluates its arguments. Strict functions are the norm in most programming languages, and indeed most languages only support functions that expect their arguments fully evaluated. Unless we tell it otherwise, any function definition in Kotlin will be strict (and all the functions we've defined so far have been strict). As an example, consider the following function:

```
fun square(x: Double): Double = x * x
```

When you invoke `square(41.0 + 1.0)`, the function `square` will receive the evaluated value of `42.0` because it's strict. If you invoke `square(exitProcess(-1))`, the program will be terminated before `square` has a chance to do anything, since the `exitProcess(-1)` expression will be evaluated before entering the body of `square`.

Although we haven't yet learned the syntax for indicating non-strictness in Kotlin, you're almost certainly familiar with the concept. For example, the short-circuiting Boolean functions `&&` and `||`, found in many programming languages including Kotlin, are non-strict. You may be used to thinking of `&&` and `||` as built-in syntax, part of the language, but you can also think of them as functions that may or may not choose not to evaluate their arguments. The function `&&` takes two `Boolean` arguments, but only evaluates the second argument if the first is `true`:

```
>>> false && { println("!!"); true }.invoke() //does not print anything
res0: kotlin.Boolean = false
```

And `||` only evaluates its second argument if the first is `false`:

```
>>> true || { println("!!"); false }.invoke() //does not print anything either
res1: kotlin.Boolean = true
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Another example of non-strictness is the `if` control construct in Kotlin:

```
val result =
    if (input.isEmpty()) exitProcess(-1) else input
```

Even though `if` is a built-in language construct in Kotlin, it can be thought of as a function accepting three parameters: a condition of type `Boolean`, an expression of some type `A` to return in the case that the condition is `true`, and another expression of the same type `A` to return if the condition is `false`. This `if` function would be non-strict, since it won't evaluate all of its arguments. To be more precise, we'd say that the `if` function is strict in its condition parameter, since it'll always evaluate the condition to determine which branch to take, and non-strict in the two branches for the `true` and `false` cases, since it'll only evaluate one or the other based on the condition.

In Kotlin, we can write non-strict functions by accepting some of our arguments unevaluated. Since Kotlin has no way of expressing unevaluated arguments, we always need to do this explicitly. Here's a non-strict `if` function:

```
fun <A> if2(
    cond: Boolean,
    onTrue: () -> A,   ❶
    onFalse: () -> A
): A = if (cond) onTrue() else onFalse()

val y = if2((a < 22),
    { println("a") },   ❷
    { println("b") }
)
```

❶ The function parameter type for a lazy value type `A` is `() -> A`

❷ The function literal syntax for creating a `() -> A`

The arguments we'd like to pass unevaluated have a `() ->` immediately before their type. A value of type `() -> A` is a function that accepts zero arguments and returns an `A`.[23] In general, the unevaluated form of an expression is called a *thunk*, and we can *force* the thunk to evaluate the expression and get a result. We do so by invoking the function, passing an empty argument list, as in `onTrue()` or `onFalse()`. Likewise, callers of `if2` have to explicitly create thunks, and the syntax follows the same conventions as the function literal syntax we've already seen. Overall, this syntax makes it very clear what's happening—we're passing a function of no arguments in place of each non-strict parameter, and then explicitly calling this function to obtain a result in the body.

With this syntax, an argument that's passed unevaluated to a function will be evaluated once for each place it's referenced in the body of the function. That is, Kotlin won't (by default) cache the result of evaluating an argument:

```
fun maybeTwice1(b: Boolean, i: () -> Int) =
    if (b) i() + i() else 0
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
>>> val x = maybeTwice(true, { println("hi"); 1 + 41 })
hi
hi
```

Here, `i` is referenced twice in the body of `maybeTwice`, and we've made it particularly obvious that it's evaluated each time by passing the block `{ println("hi"); 1+41 }`, which prints `hi` as a side effect before returning a result of `42`. The expression `1 + 41` will be computed twice as well. We can cache the value explicitly if we wish to only evaluate the result once, by delegating to the `lazy` builtin function on assigning a new value:

```
fun maybeTwice2(b: Boolean, i: () -> Int) {
    val j: Int by lazy(i)
    if (b) j + j else 0
}
```

```
>>> val x = maybeTwice2(true, { println("hi"); 1 + 41 })
hi
```

We use *lazy evaluation* to initialize the value of `j`. This will defer initialization until it is referenced by the `if` statement. It will will also cache the result so that subsequent references to this value don't trigger repeated evaluation. This mechanism used for evaluation is not vital to this discussion, but is treated in more detail in the sidebar titled *Lazy initialization*.

SIDEBAR **Formal definition of strictness**

If the evaluation of an expression runs forever or throws an error instead of returning a definite value, we say that the expression doesn't *terminate*, or that it evaluates to *bottom*. A function `f` is *strict* if the expression `f(x)` evaluates to bottom for all `x` that evaluate to bottom.

NOTE We say that non-strict function arguments are passed in *by name*, where strict arguments are passed in *by value*.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Lazy initialization**

Lazy initialization is the tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. The full definition can be found here on Wikipedia: en.wikipedia.org/wiki/Lazy_initialization

This language feature is implemented in Kotlin by way of a builtin function called `lazy`. An instance of `Lazy<T>` (with `T` being the type of the value to be assigned) is returned when this function is invoked with a lambda thunk as argument. This `Lazy` object serves as delegate for implementing a lazy property. Delegation is expressed by using the `by` keyword.

```
val x: Int by lazy { expensiveOp() }   ❶

fun useit() =
    if (x > 10) "hi"   ❷
    else if (x == 0) "zero"   ❸
    else ("lo")
```

❶ The `by` keyboard is used to bind the `Lazy<Int>` returned by `lazy` to `x`

❷ When `x` is evaluated in the conditional statement, the call to `expensiveOp` will be made and the result cached

❸ Cached value is used instead of making another call to `expensiveOp`

The lazy property `x` is initialized on first access by executing `expensiveOp` inside the lambda thunk that was passed into the `lazy` function. The result is then cached inside the delegate object for subsequent evaluations to take advantage of.

Access to the thunk is thread-safe by default using concurrent locks, but the behavior can be altered by using different lazy thread-safe modes. The use of these modes is beyond the scope of this sidebar, but is well documented on the Kotlin website[24].

5.2 An extended example: lazy lists

Let's return to the problem posed at the beginning of this chapter, where we were performing several transformations on a list that required multiple traversals. We'll explore how laziness can be used to improve the efficiency and modularity of functional programs using *lazy lists*, or *streams*, as an example. We'll see how chains of transformations on streams are fused into a single pass through the use of laziness. Here's a simple `Stream` definition. There are a few new things here we'll discuss next.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 5.2 Definition for the `Stream` data type with its sealed implementations

```
sealed class Stream<out A> {

    companion object {

        //smart constructors
    }
}

data class Cons<out A>(
    val head: () -> A,
    val tail: () -> Stream<A>
) : Stream<A>()

object Empty : Stream<Nothing>()
```

This type looks identical to our `List` type, except that the `Cons` data constructor takes *explicit* thunks (`() -> A` and `() -> Stream<A>`) instead of regular strict values. If we wish to examine or traverse the `Stream`, we need to force these thunks as we did earlier in our definition of `if2`. For instance, here's an extenstion function to optionally extract the head of a `Stream`:

```
fun <A> Stream<A>.headOption(): Option<A> =
    when (this) {
        is Empty -> None
        is Cons -> Some(head())   ❶
    }
```

❶ Explicit forcing of the `head` thunk using `head()`

As we are adding behaviour to a `Stream` instance, we have the `head` and `tail` values available in `this` when smart-cast to `Cons` in the `when` construct. Note that we have to force `head` explicitly via `head()`, but other than that, the code works the same way as it would for `List`. But this ability of `Stream` to evaluate only the portion actually demanded (we don't evaluate the tail of the `Cons`) is very useful, as we'll see in the following section.

5.2.1 Memoizing of streams and avoiding recomputation

Evaluations representing expensive computations should be avoided at all costs. One way of preventing excessive evaluation is a technique called *memoization*. In applying this technique, we prevent multiple evaluations of expensive computations by caching the result of the initial evaluation. The net result is that every pure expression is only evaluated once and then reused for the remainder of that program.

We typically want to cache the values of a `Cons` node, once they are forced. If we use the `Cons` data constructor directly, for instance, this code will actually compute `expensive(y)` twice:

```
val x = Cons({ expensive(y) }, { tl })
val h1 = x.headOption()
val h2 = x.headOption()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

We typically avoid this problem by defining *smart* constructors, which is what we call a function for constructing a data type that ensures some additional invariant or provides a slightly different signature than the "real" constructors. By convention, smart constructors live in the companion object of the base class and their names typically lowercase the first letter of the corresponding data constructor. Here, our `cons` smart constructor takes care of memoizing the by-name arguments for the head and tail of the `Cons`. This is a common trick, and it ensures that our thunk will only do its work once, when forced for the first time. Subsequent forces will return the cached lazy val:

```
companion object {

    //smart constructors
    fun <A> cons(hd: () -> A, tl: () -> Stream<A>): Stream<A> {
        val head: A by lazy(hd)
        val tail: Stream<A> by lazy(tl)
        return Cons({ head }, { tail })
    }

    fun <A> empty(): Stream<A> = Empty
}
```

The `empty` smart constructor just returns `Empty`, but annotates `Empty` as a `Stream<A>`, which is better for type inference in some cases.[25] We can see how both smart constructors are used in the `Stream.of` function:

```
companion object {

    //smart constructors

    fun <A> of(vararg xs: A): Stream<A> =
        if (xs.isEmpty()) empty()
        else cons({ xs[0] },
            { of(*xs.sliceArray(1 until xs.size)) })
}
```

Since Kotlin does not take care of wrapping the arguments to `cons` in thunks, we need to do this explicitly by surrounding `xs[0]` and `of(*xs.sliceArray(1 until xs.size))` in lambdas so that the expressions won't be evaluated until we force the `Stream`.

5.2.2 Helper functions for inspecting streams

Before continuing, let's write a few helper functions to make inspecting streams easier.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 5.1

Write a function to convert a `Stream` to a `List`, which will force its evaluation to let you look at the result in the REPL. You can convert to the singly-linked `List` type that we developed in chapter 3, and you can implement this and other functions that operate on a `Stream` using extension methods.

```
fun <A> Stream<A>.toList(): List<A> = TODO()
```

TIP Think about stack safety when implementing this function. Consider tail call elimination and the use of another method that you implemented on `List`.

EXERCISE 5.2

Write the function `take(n)` for returning the first `n` elements of a `Stream`, and `drop(n)` for skipping the first `n` elements of a `Stream`.

```
fun <A> Stream<A>.take(n: Int): Stream<A> = TODO()

fun <A> Stream<A>.drop(n: Int): Stream<A> = TODO()
```

EXERCISE 5.3

Write the function `takeWhile` for returning all starting elements of a `Stream` that match the given predicate.

```
fun <A> Stream<A>.takeWhile(p: (A) -> Boolean): Stream<A> = TODO()
```

TIP You can use `take` and `toList` together to inspect streams in the REPL. For example, try printing `Stream.of(1,2,3).take(2).toList()`. This is also of great use during assertion expressions in unit tests.

5.3 Separating program description from evaluation

A major theme in functional programming is *separation of concerns*. We want to separate the description of computations from actually running them. We've already touched on this theme in previous chapters in different ways. For example, first-class functions capture some computation in their bodies but only execute it once they receive their arguments. And we used `Option` to capture the fact that an error occurred, where the decision of what to do about it became a separate concern. With `Stream`, we're able to build up a computation that produces a sequence of elements without running the steps of that computation until we actually need those elements.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Laziness lets us separate the *description* of an expression from the *evaluation* of that expression. This gives us a powerful ability—we may choose to describe a "larger" expression than we need, and then evaluate only a portion of it. As an example, let's look at the function `exists` that checks whether an element matching a `Boolean` function exists in this `Stream`:

```
fun <A> Stream<A>.exists(p: (A) -> Boolean): Boolean =
    when (this) {
        is Cons -> p(this.head()) || this.tail().exists(p)
        else -> false
    }
```

Note that `||` is non-strict in its second argument. If `p(head())` returns `true`, then `exists` terminates the traversal early and returns `true` as well. Remember also that the tail of the stream is a lazy `val`. So not only does the traversal terminate early, the tail of the stream is never evaluated at all! So whatever code would have generated the tail is never actually executed.

The `exists` function here is implemented using explicit recursion. But remember that with `List` in chapter 3, we could implement a general recursion in the form of `foldRight`. We can do the same thing for `Stream`, but lazily:

Listing 5.3 The `foldRight` function on `Stream` is used to generalize recursion.

```
fun <A, B> Stream<A>.foldRight(
    z: () -> B,
    f: (A, () -> B) -> B
): B =  ❶
    when (this) {
        is Cons -> f(this.head()) { tail().foldRight(z, f) }  ❶
        else -> z()
    }
```

❶ The type `() -> B` means that the function `f` takes its second argument by name and may choose not to evaluate it.

❷ If `f` doesn't evaluate its second argument, the recursion never occurs.

This looks very similar to the `foldRight` we wrote for `List` in chapter 3, but note how our combining function `f` is non-strict in its second parameter. If `f` chooses not to evaluate its second parameter, the traversal will be terminated early. We can see this by using `foldRight` to implement `exists2`:[26]

```
fun <A> Stream<A>.exists2(p: (A) -> Boolean): Boolean =
    foldRight({ false }, { a, b -> p(a) || b() })
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 5.4

Implement `forAll`, which checks that all elements in the `Stream` match a given predicate. Your implementation should terminate the traversal as soon as it encounters a non-matching value.

```
fun <A> Stream<A>.forAll(p: (A) -> Boolean): Boolean = TODO()
```

EXERCISE 5.5

Use `foldRight` to implement `takeWhile`.

EXERCISE 5.6

Implement `headOption` using `foldRight`.

EXERCISE 5.7

Implement `map`, `filter`, `append`, and `flatMap` using `foldRight`. The `append` method should be non-strict in its argument.

TIP Consider using previously defined methods where applicable.

Note that these implementations are *incremental*—they don't fully generate their answers. It's not until some other computation looks at the elements of the resulting `Stream` that the computation to generate that `Stream` actually takes place—and then it will do just enough work to generate the requested elements. Because of this incremental nature, we can call these functions one after another without fully instantiating the intermediate results.

Let's look at a simplified program trace for a fragment of the motivating example we started this chapter with, `Stream.of(1, 2, 3, 4).map { it + 10 }.filter { it % 2 == 0 }`. We leave off the final transformation, `.map { it * 3 }` for the sake of simplicity. We'll convert this expression to a `List` to force evaluation. Take a minute to work through this trace to understand what's happening. It's a bit more challenging than the trace we looked at earlier in this chapter. Remember, a trace like this is just the same expression over and over again, evaluated by one more step each time.

Listing 5.4 Trace representing evaluation order of operations on a `Stream`

```
import chapter3.Cons as ConsL
import chapter3.Nil as NilL
```

```
        Stream.of(1, 2, 3, 4).map { it + 10 }
            .filter { it % 2 == 0 }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        .map { it * 3 }.toList()

    cons({ 11 }, { Stream.of(2, 3, 4).map { it + 10 } })
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList()   ❶

    Stream.of(2, 3, 4).map { it + 10 }
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList()   ❶

    cons({ 12 }, { Stream.of(3, 4).map { it + 10 } })
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList()   ❷

    ConsL(36, Stream.of(3, 4).map { it + 10 }
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList())   ❸

    ConsL(36, cons({ 13 }, { Stream.of(4).map { it + 10 } })
            .filter { it % 2 == 0 }
            .map { it * 3 }.toList())   ❺

    ConsL(36, Stream.of(4).map { it + 10 }
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList())   ❻

    ConsL(36, cons({ 14 }, { Stream.empty<Int>().map { it + 10 } })
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList())   ❼

    ConsL(36, ConsL(42, Stream.empty<Int>().map { it + 10 }
        .filter { it % 2 == 0 }
        .map { it * 3 }.toList()))   ❽

    ConsL(36, ConsL(42, NilL))   ❾
```

❶ Apply `map` to the first element

❷ Apply `filter` to the first element; predicate returns `false`

❸ Apply `map` to the second element

❹ Apply `filter` to the second element; predicate returns `true`; apply second `map`; produces first element of result

❺ Apply `map` to the third element

❻ Apply `filter` to the third element; predicate returns `false`

❼ Apply `map` to the last element

❽ Apply `filter` to the last element; predicate returns `true`; apply second `map`; produces second element of result

❾ End of stream `Empty` has been reached. Now `map` and `filter` have no more work to do; empty stream becomes `Nil`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

NOTE Kotlin features the use of import aliasing using the `as` keyword. This allows for the renaming of objects, classes, methods and the like to be imported with a different name. In this case the `Cons` is already defined in `Stream`, so an import alias for the `List` data type's `Cons` and `Nil` are imported as `ConsL` and `NilL` respectively. This is a handy trick for when namespace clashes occur without resorting to full package name qualifiers.

```
import chapter3.Cons as ConsL
import chapter3.Nil as NilL
```

The thing to notice in this trace is how the `filter` and `map` transformations are interleaved—the computation alternates between generating a single element of the output of `map`, and testing with `filter` to see if that element is divisible by `2` (adding it to the output list if it is). Note that we don't fully instantiate the intermediate stream that results from the `map`. It's exactly as if we had interleaved the logic using a special-purpose loop. For this reason, people sometimes describe streams as "first-class loops" whose logic can be combined using higher-order functions like `map` and `filter`.

Since intermediate streams aren't instantiated, it's easy to reuse existing combinators in novel ways without having to worry that we're doing more processing of the stream than necessary. For example, we can reuse `filter` to define `find`, a method to return just the first element that matches if it exists. Even though `filter` transforms the whole stream, that transformation is done lazily, so `find` terminates as soon as a match is found:

```
fun <A> Stream<A>.find(p: (A) -> Boolean): Option<A> =
    filter(p).headOption()
```

The incremental nature of stream transformations also has important consequences for memory usage. Because intermediate streams aren't generated, a transformation of the stream requires only enough working memory to store and transform the current element. For instance, in the transformation `Stream.of(1, 2, 3, 4).map { it + 10 }.filter { it % 2 == 0 }`, the garbage collector can reclaim the space allocated for the values `11` and `13` emitted by `map` as soon as `filter` determines they aren't needed. Of course, this is a simple example; in other situations we might be dealing with larger numbers of elements, and the stream elements themselves could be large objects that retain significant amounts of memory. Being able to reclaim this memory as quickly as possible can cut down on the amount of memory required by your program as a whole.

We'll have a lot more to say about defining memory-efficient streaming calculations, in particular calculations that require I/O, in part 4 of this book.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

5.4 Producing infinite data streams through corecursive functions

The functions we've written also work for *infinite streams* because they're incremental. Here's an example of an infinite `Stream` of `1` s:

```
fun ones(): Stream<Int> = Stream.cons({ 1 }, { ones() })
```

Although `ones` is infinite, the functions we've written so far only inspect the portion of the stream needed to generate the demanded output. For example:

```
>>> ones().take(5).toList()
res0: chapter3.List<kotlin.Int> = Cons(head=1, tail=Cons(head=1,
                                                         tail=Cons(head=1,
                                                         tail=Cons(head=1,
                                                         tail=Cons(head=1, tail=Nil)))))

>>> ones().exists { it % 2 != 0 }
res1: Boolean = true
```

Figure 5.1 The ones function is incremental, producing an infinite stream of 1 values on demand.

Try playing with a few other examples:

```
ones().map { it + 1 }.exists { it % 2 == 0 }
ones().takeWhile { it == 1 }
ones().forAll { it == 1 }
```

In each case, we get back a result immediately. Be careful though, since it's easy to write expressions that never terminate or aren't stack-safe. For example, `ones.forAll { it != 1 }`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

will forever need to inspect more of the series since it'll never encounter an element that allows it to terminate with a definite answer (this will manifest as a stack overflow rather than an infinite loop).[27]

Let's see what other functions we can discover for generating streams.

EXERCISE 5.8

Generalize `ones` slightly to the function `constant`, which returns an infinite `Stream` of a given value.

```
fun <A> constant(a: A): Stream<A> = TODO()
```

EXERCISE 5.9

Write a function that generates an infinite stream of integers, starting from `n`, then `n + 1`, `n + 2`, and so on.[28]

```
fun from(n: Int): Stream<Int> = TODO()
```

EXERCISE 5.10

Write a function `fibs` that generates the infinite stream of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, and so on.

```
fun fibs(): Stream<Int> = TODO()
```

EXERCISE 5.11

Write a more general stream-building function called `unfold`. It takes an initial state, and a function for producing both the next state and the next value in the generated stream.

```
fun <A, S> unfold(z: S, f: (S) -> Option<Pair<A, S>>): Stream<A> = TODO()
```

`Option` is used to indicate when the `Stream` should be terminated, if at all. The function `unfold` is a very general `Stream`-building function.

The `unfold` function is an example of what's sometimes called a *corecursive* function. Whereas a recursive function consumes data, a corecursive function *produces* data. And whereas recursive functions terminate by recursing on smaller inputs, corecursive functions need not terminate so long as they remain *productive*, which just means that we can always evaluate more of the result in a finite amount of time. The `unfold` function is productive as long as `f` terminates, since we just need to run the function `f` one more time to generate the next element of the `Stream`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Corecursion is also sometimes called guarded *recursion*, and productivity is also sometimes called *cotermination*. These terms aren't that important to our discussion, but you'll hear them used sometimes in the context of functional programming. If you're curious to learn where they come from and understand some of the deeper connections, follow the references in the chapter notes.

EXERCISE 5.12

Write `fibs`, `from`, `constant`, and `ones` in terms of `unfold`.[29]

EXERCISE 5.13

Use `unfold` to implement `map`, `take`, `takeWhile`, `zipWith` (as in chapter 3) and `zipAll`. The `zipAll` function should continue the traversal as long as either stream has more elements—it uses `Option` to indicate whether each stream has been exhausted.

```
fun <A, B> Stream<A>.map(f: (A) -> B): Stream<B> = TODO()

fun <A> Stream<A>.take(n: Int): Stream<A> = TODO()

fun <A> Stream<A>.takeWhile(p: (A) -> Boolean): Stream<A> = TODO()

fun <A, B, C> Stream<A>.zipWith(
    that: Stream<B>,
    f: (A, B) -> C
): Stream<C> = TODO()

fun <A, B> Stream<A>.zipAll(
    that: Stream<B>
): Stream<Pair<Option<A>, Option<B>>> = TODO()
```

Now that we have some practice writing stream functions, let's return to the exercise we covered at the end of chapter 3—a function, `hasSubsequence`, to check whether a list contains a given subsequence. With strict lists and list-processing functions, we were forced to write a rather tricky monolithic loop to implement this function without doing extra work. Using lazy lists, can you see how you could implement `hasSubsequence` by combining some other functions we've already written? Try to ponder this on your own before continuing.

EXERCISE 5.14 (Hard)

Implement `startsWith` using functions you've written. It should check if one `Stream` is a prefix of another. For instance, `Stream(1,2,3) startsWith Stream(1,2)` would be `true`.

```
fun <A> Stream<A>.startsWith(that: Stream<A>): Boolean = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP This can be solved by re-using only functions developed with `unfold` earlier in this chapter.

EXERCISE 5.15

Implement `tails` using `unfold`. For a given `Stream`, `tails` returns the `Stream` of suffixes of the input sequence, starting with the original `Stream`. For example, given `Stream.of(1,2,3)`, it would return `Stream.of(Stream.of(1,2,3), Stream.of(2,3), Stream.of(3), Stream.empty())`.

```
fun <A> Stream<A>.tails(): Stream<Stream<A>> = TODO()
```

We can now implement `hasSubsequence` using functions we've already written:

```
fun <A> Stream<A>.hasSubsequence(s: Stream<A>): Boolean =
    this.tails().exists { it.startsWith(s) }
```

This implementation performs the same number of steps as a more monolithic implementation using nested loops with logic for breaking out of each loop early. By using laziness, we can compose this function from simpler components and still retain the efficiency of the more specialized (and verbose) implementation.

EXERCISE 5.16 (Hard/Optional)

Generalize `tails` to the function `scanRight`, which is like a `foldRight` that returns a stream of the intermediate results. For example:

```
>>> Stream.of(1, 2, 3).scanRight(0, { a, b -> a + b }).toList()
res1: chapter3.List<kotlin.Int> = Cons(head=6, tail=Cons(head=5,
                                                         tail=Cons(head=3,
                                                         tail=Cons(head=0,
                                                         tail=Nil))))
```

This example should be equivalent to the expression `List.of(1+2+3+0, 2+3+0, 3+0, 0)`. Your function should reuse intermediate results so that traversing a `Stream` with `n` elements always takes time linear in `n`. Can it be implemented using `unfold`? How, or why not? Could it be implemented using another function we've written?

In this chapter, we introduced non-strictness as a fundamental technique for implementing efficient and modular functional programs. Non-strictness can be thought of as a technique for recovering some efficiency when writing functional code, but it's also a much bigger idea—non-strictness can improve modularity by separating the description of an expression from the how-and-when of its evaluation. Keeping these concerns separate lets us reuse a description

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

in multiple contexts, evaluating different portions of our expression to obtain different results. We weren't able to do that when description and evaluation were intertwined as they are in strict code. We saw a number of examples of this principle in action over the course of the chapter, and we'll see many more in the remainder of the book.

We'll switch gears in the next chapter and talk about purely functional approaches to state. This is the last building block needed before we begin exploring the process of functional design.

5.5 Summary

- Strict expressions are evaluated at the moment that they are bound to a variable. This is acceptable for simple expressions, but not for expensive computations which should be deferred as long as possible.
- *Non-strict*, or *lazy* evaluation results in computations being deferred to the point where the value is first referenced. This allows expensive computations to be evaluated on demand.
- A *thunk* is the unevaluated form of an expression, and is a humorous past particle of "think".
- Lazy initialization can be achieved by wrapping an expression in a *thunk*, which can be forced to execute explicitly at a later stage if required.
- You can use the `Stream` data type to model a lazy list implementation using `Cons` and `Empty` sealed types.
- *Memoizing* is a technique used to prevent multiple evaluations of an expression by caching the result of the first evaluation.
- A *smart* constructor provides a function with a slightly different signature to the actual constructor. It ensures some additional invariant to what the original offers.
- We are able to separate the concerns of *description* and *evaluation* when applying laziness, resulting in the ability to describe a larger expression than needed while only evaluating a smaller portion.
- Infinite streams can be generated using *corecursive* functions to produce data incrementally. The `unfold` function is such a stream generator.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

6 Purely functional state

This chapter covers:

- Writing pure stateful APIs by making state updates explicit
- Identifying general repetition when dealing with pure state transitions
- Using combinators to abstract over explicit state transitions
- Combining both multiple and nested state actions
- Introducing a general state action data type
- Using an imperative style when dealing with state actions

Working with program state is tricky, and even more so in the context of functional programming where we value principals such as immutability and the eradication of side effects. Mutating state comes at a huge cost, making programs difficult to reason about and maintain. Fortunately we have a design pattern at hand to deal with program state in a pure functional way. Applying this pattern allows us to deal with state in a deterministic fashion and subsequently allows us to reason about and test our programs with greater ease.

By viewing the program state as a *transition* or *action* that is passed along as context during a series of transformations, we can contain and localize the complexity that is usually associated with state machines. We can even take this a step further by *hiding* these state transitions altogether through the use of higher order combinator functions that pass state actions along implicitly in the background. A pattern begins to emerge as we combine these concepts of passing and hiding state transitions in the background.

In this chapter, we'll see how to write purely functional programs that manipulate state, using the simple domain of *random number generation* as the example. Although by itself it's not the most compelling use case for the techniques in this chapter, the simplicity of random number generation makes it a good first example. We'll see more compelling use cases in parts 3 and 4

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

of the book, especially part 4, where we'll say a lot more about dealing with state and effects. The goal here is to give you the basic pattern for how to make *any* stateful API purely functional. As you start writing your own functional APIs, you'll likely run into many of the same questions that we'll explore here.

6.1 Generating random numbers using side effects

Let's begin by looking at the contrived example of generating random numbers using a pseudo-random number generator. This action would usually be handled using mutable state and side effects. Let's demonstrate this using a typical imperative solution before we move on to show how this can be achieved in a pure functional way.

If you need to generate random numbers[30] in Kotlin, there's a class in the standard library, `kotlin.random.Random`,[31] with a pretty typical imperative API that relies on side effects. Listing 6.1 is an example of the `Random` class usage:

Listing 6.1 Using the Kotlin `Random` class to demonstrate mutation of internal state.

```
>>> val rng = kotlin.random.Random   ❶

>>> rng.nextDouble()
res1: kotlin.Double = 0.2837830961138915

>>> rng.nextDouble()
res2: kotlin.Double = 0.7994579111535903

>>> rng.nextInt()
res3: kotlin.Int = -1630636086

>>> rng.nextInt(10)
res4: kotlin.Int = 8   ❷
```

❶ Creates a new random number generator seeded with the current system time

❷ Gets a random integer between 0 and 9

Even if we didn't know anything about what happens inside `kotlin.random.Random`, we can assume that the object `rng` has some internal state that gets updated after each invocation, since we'd otherwise get the same value each time we called `nextInt` or `nextDouble`. These methods aren't referentially transparent because the state updates are performed as a side effect. As we know from our discussion in chapter 1, this implies that they aren't as testable, composable, modular, and easily parallelized as they could be.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 6.1 Using the Random class to generate pseudo-random numbers while mutating state

Let's just take testability as an example. If we want to write a method that makes use of randomness, we need tests to be reproducible. Let's say we had the following side-effecting method, intended to simulate the rolling of a single six-sided die, which *should* return a value between 1 and 6, inclusive:

Listing 6.2 A method simulating the roll of a die with an off-by-one error.

```
fun rollDie(): Int {    ❶
    val rng = kotlin.random.Random
    return rng.nextInt(6)    ❷
}
```

❶ Method should return a random number from 1 to 6

❷ Generator returns a random number from 0 to 5

This method has an off-by-one error. It's supposed to return a value between 1 and 6, but it actually returns a value from 0 to 5. But even though it doesn't work properly, five out of six times a test of this method will meet the test specification! And if a test did fail, it would be ideal if we could reliably reproduce the failure before we attempt to fix it.

Note that what's important here is not this specific example, but the general idea. In this case, the bug is obvious and easy to reproduce. But we can easily imagine a situation where the method is much more complicated and the bug far more subtle. The more complex the program and the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

subtler the bug, the more important it is to be able to reproduce bugs in a reliable way.

One suggestion for making such a test more deterministic might be to pass in the random number generator. That way, when we want to reproduce a failed test, we can pass the *same* generator that caused the test to fail:

```
fun rollDie2(rng: Random): Int = rng.nextInt(6)
```

But there's a problem with this solution. The "same" generator has to be both created with the same seed, and also be in the same state, which means that its methods have been called a certain number of times since it was created. That will be really difficult to guarantee, because every time we call `nextInt`, for example, the previous state of the random number generator is destroyed. Do we now need a separate mechanism to keep track of how many times we've called the methods on `Random`?

No! The answer to all of this, of course, is that we should eschew side effects on principle!

6.2 Purely functional random number generation

Let's evolve our design by removing this undesirable side effect. We will do so by reworking our example into a pure functional solution to recover referential transparency. We can do so by making the state updates *explicit*. Don't update the state as a side effect, but simply return the new state along with the value that we're generating. Listing 6.3 shows one possible interface to a random number generator.

Listing 6.3 Interface to a random number generator

```
interface RNG {
    fun nextInt(): Pair<Int, RNG>
}
```

This method should generate a random `Int`. We'll later define other functions in terms of `nextInt`. Rather than returning only the generated random number (as is done in `kotlin.random.Random`) and updating some internal state by *mutating* it in place, we return the random number *and* the new state, leaving the old state unmodified.[32] In effect, we separate the concern of *computing* what the next state is from the concern of *communicating* the new state to the rest of the program. No global mutable memory is being used—we simply return the next state back to the caller. This leaves the caller of `nextInt` in complete control of what to do with the new state. Note that we're still *encapsulating* the state, in the sense that users of this API don't know anything about the implementation of the random number generator itself.

For our example, we do need to have an implementation so let's pick a simple one. The following is an algorithm called a *linear congruential generator* (bit.ly/2MtQsgc). The details of this implementation aren't really that important, but notice that `nextInt` returns both the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

generated value and a new `RNG` to use for generating the next value.

Listing 6.4 A purely functional random number generator, implementing `RNG`

```
data class SimpleRNG(val seed: Long) : RNG {
    override fun nextInt(): Pair<Int, RNG> {
        val newSeed =
            (seed * 0x5DEECE66DL + 0xBL) and
                0xFFFFFFFFFFFFL   ❶
        val nextRNG = SimpleRNG(newSeed)   ❷
        val n = (newSeed ushr 16).toInt()   ❸
        return Pair(n, nextRNG)   ❹
    }
}
```

❶ We use the current seed to generate a new seed. `and` is a bitwise AND.

❷ The next state, which is an `RNG` instance created from the new seed.

❸ The value `n` is the new pseudo-random integer. `ushr` is a right binary shift with zero fill.

❹ The return value is a `Pair<Int, RNG>`, containing both a pseudo-random integer and the next RNG state.

Listing 6.5 demonstrates how we would use this API from the interpreter.

Listing 6.5 Demonstrating repeatable random number generation using `SimpleRNG` in the REPL

```
>>> val rng = SimpleRNG(42)   ❶

>>> val (n1, rng2) = rng.nextInt()   ❶

>>> println("n1:$n1; rng2:$rng2")

n1:16159453; rng2:SimpleRNG(seed=1059025964525)

>>> val (n2, rng3) = rng2.nextInt()   ❸

>>> println("n2:$n2; rng3:$rng3")

n2:-1281479697; rng3:SimpleRNG(seed=197491923327988)
```

❶ Choose an arbitrary value to initialize `SimpleRNG`

❷ Destructure the `Pair<Int, RNG>` returned from `nextInt`

We can run this sequence of statements as many times as we want and we'll always get the same values. When we call `rng.nextInt()`, it will always return `16159453` and a new `RNG`, whose `nextInt` will always return `-1281479697`. In other words, we have now arrived at a pure API.

6.3 Making stateful APIs pure

This problem of making seemingly stateful APIs pure and its solution of having the API *compute* the next state rather than actually mutate anything aren't unique to random number generation. This problem comes up frequently, and we can always deal with it in this same way.[33]

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

For instance, suppose you have a data repository that can produce a sequence of numbers.

```
class MutatingSequencer {
    private var repo: Repository = TODO()
    fun nextInt(): Int = TODO()
    fun nextDouble(): Double = TODO()
}
```

Now suppose `nextInt` and `nextDouble` each mutate `repo` in some way. We can mechanically translate this interface to a purely functional one by making the state transition explicit.

```
interface StateActionSequencer {
    fun nextInt(): Pair<Int, StateActionSequencer>
    fun nextDouble(): Pair<Double, StateActionSequencer>
}
```

Whenever we use this pattern, we make the caller responsible for passing the next computed state through the rest of the program. Going back to the pure `RNG` interface shown earlier in Listing 6.3, if we reuse a previous `RNG`, it will always generate the same value it generated before.

```
fun randomPair(rng: RNG): Pair<Int, Int> {
    val (i1, _) = rng.nextInt()
    val (i2, _) = rng.nextInt()
    return Pair(i1, i2)
}
```

Here `i1` and `i2` will be the same! If we want to generate two distinct numbers, we need to use the `RNG` returned by the first call to `nextInt` to generate the second `Int`:

Listing 6.6 Using different `RNG` instances to generate subsequent random numbers.

```
fun randomPair2(rng: RNG): Pair<Pair<Int, Int>, RNG> {
    val (i1, rng2) = rng.nextInt()
    val (i2, rng3) = rng2.nextInt()   ❶
    return Pair(Pair(i1, i2), rng3)   ❶
}
```

❶ Use `rng2` instead of `rng` here.

❷ Return final state `rng3` after generating random numbers, allows caller to continue generating random values.

You can see the general pattern, and perhaps you can also see how it might get tedious to use this API directly. Let's write a few functions to generate random values and see if we notice any repetition that we can factor out.

EXERCISE 6.1

Write a function that uses `RNG.nextInt` to generate a random integer between `0` and `Int.MAX_VALUE` (inclusive).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP Each negative value must be mapped to a distinct non-negative value. Make sure to handle the corner case when `nextInt` returns `Int.MIN_VALUE`, which doesn't have a non-negative counterpart.

EXERCISE

```
fun nonNegativeInt(rng: RNG): Pair<Int, RNG> = TODO()
```

SIDEBAR **Dealing with awkwardness in functional programming**

As you write more functional programs, you'll sometimes encounter situations where the functional way of expressing a program feels awkward or tedious. Does this imply that purity is the equivalent of trying to write an entire novel without using the letter *E*? Of course not! Awkwardness like this is almost always a sign of some missing abstraction waiting to be discovered.

When you encounter these situations, we encourage you to plow ahead and look for common patterns that you can factor out. Most likely, this is a problem that others have encountered, and you may even rediscover the "standard" solution yourself. Even if you get stuck, struggling to puzzle out a clean solution yourself will help you to better understand what solutions others have discovered to deal with similar problems.

With practice, experience, and more familiarity with the idioms contained in this book, expressing a program functionally will become effortless and natural. Of course, good design is still hard, but programming using pure functions greatly simplifies the design space.

EXERCISE **6.2**

Write a function to generate a `Double` between `0` and `1`, not including `1`. In addition to the function you already developed, you can use `Int.MAX_VALUE` to obtain the maximum positive integer value, and you can use `x.toDouble()` to convert an `x: Int` to a `Double`.

```
fun double(rng: RNG): Pair<Double, RNG> = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 6.3

Write functions to generate a `Pair<Int, Double>`, a `Pair<Double, Int>`, and a `Triple<Double, Double, Double>`. You should be able to reuse the functions you've already written.

```
fun intDouble(rng: RNG): Pair<Pair<Int, Double>, RNG> = TODO()

fun doubleInt(rng: RNG): Pair<Pair<Double, Int>, RNG> = TODO()

fun double3(rng: RNG): Pair<Triple<Double, Double, Double>, RNG> =
    TODO()
```

EXERCISE 6.4

Write a function to generate a list of random integers.

```
fun ints(count: Int, rng: RNG): Pair<List<Int>, RNG> = TODO()
```

6.4 An implicit approach to passing state actions

Up to this point we have moved from an approach using mutable state to a pure functional way of propagating state explicitly, thereby avoiding side effects. Apart from being procedural and error-prone, passing this state along feels unnecessarily cumbersome and tedious. Let's evolve our design even further by removing the necessity of passing this state along explicitly.

Looking back at our implementations, we'll notice a common pattern: each of our functions has a type of the form `(RNG) -> Pair<A, RNG>` for some type `A`. Functions of this type are called *state actions* or *state transitions* because they transform `RNG` states from one to the next. These state actions can be combined using *combinators*, which are higher-order functions that we'll define in this section. Since it's pretty tedious and repetitive to pass the state along ourselves, we want our combinators to pass the state from one action to the next automatically.

To make the type of actions convenient to talk about, and to simplify our thinking about them, let's make a type alias for the `RNG` state action data type:

Listing 6.7 Type alias representing a state transition

```
typealias Rand<A> = (RNG) -> Pair<A, RNG>
```

We can think of a value of type `Rand<A>` as "a randomly generated `A`," although that's not really precise. It's really a state action—a *program* that depends on some `RNG`, uses it to generate an `A`, and also transitions the `RNG` to a new state that can be used by another action later.

We can now turn methods such as `RNG.nextInt`, which returns a `Pair<Int, RNG>` containing a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

generated `Int` along with the next `RNG` into values of this new type:

```
val intR: Rand<Int> = { rng -> rng.nextInt() }
```

We want to write combinators that let us combine `Rand` actions while avoiding explicitly passing along the `RNG` state. We'll end up with a kind of domain-specific language that does all of the passing for us. For example, a simple `RNG` state transition is the `unit` action, which passes the `RNG` state through without using it, always returning a constant value rather than a random value:

Listing 6.8 The `unit` combinator passes state while setting a constant

```
fun <A> unit(a: A): Rand<A> = { rng -> Pair(a, rng) }
```

There's also `map` for transforming the output of a state action without modifying the state itself. Remember, `Rand<A>` is just a type alias for a function type `(RNG) -> Pair(A, RNG)`, so this is just a kind of function composition:

Listing 6.9 The `map` combinator modifies output without modifying the state

```
fun <A, B> map(s: Rand<A>, f: (A) -> B): Rand<B> =
    { rng ->
        val (a, rng2) = s(rng)
        Pair(f(a), rng2)
    }
```

As an example of how `map` is used, here's `nonNegativeEven`, which reuses `nonNegativeInt` to generate an `Int` that's greater than or equal to zero and divisible by two:

```
fun nonNegativeEven(): Rand<Int> =
    map(::nonNegativeInt) { it - (it % 2) }
```

EXERCISE 6.5

Use `map` to reimplement `double` in a more elegant way. See exercise 6.2.

```
val doubleR: Rand<Double> = TODO()
```

6.4.1 More power by combining state actions

We've been developing an API for working with single state actions by hiding their transitions. Sometimes we need to harness more power from *multiple* state actions at once, while retaining the ability to hide their transitions in the background. We would like to keep following this approach when implementing `intDouble` and `doubleInt` from Exercise 6.3, but `map` simply doesn't have the capability to do this. What we need is a new combinator, `map2` that can combine two `RNG` actions into one using a binary rather than unary function.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 6.6

Write the implementation of `map2` based on the following signature. This function takes two actions, `ra` and `rb`, and a function `f` for combining their results, and returns a new action that combines them:

```
fun <A, B, C> map2(
    ra: Rand<A>,
    rb: Rand<B>,
    f: (A, B) -> C
): Rand<C> = TODO()
```

We only have to write the `map2` combinator once, and then we can use it to combine arbitrary `RNG` state actions. For example, if we have an action that generates values of type `A` and an action to generate values of type `B`, then we can combine them into one action that generates pairs of both `A` and `B`:

```
fun <A, B> both(ra: Rand<A>, rb: Rand<B>): Rand<Pair<A, B>> =
    map2(ra, rb) { a, b -> Pair(a, b) }
```

We can use `both` to reimplement `intDouble` and `doubleInt` from exercise 6.3 more succinctly. We do this by using the `Rand` values `intR` and `doubleR` developed earlier in the chapter:

```
val intR: Rand<Int> = { rng -> rng.nextInt() }

val doubleR: Rand<Double> =
    map(::nonNegativeInt) { i ->
        i / (Int.MAX_VALUE.toDouble() + 1)
    }

val intDoubleR: Rand<Pair<Int, Double>> = both(intR, doubleR)

val doubleIntR: Rand<Pair<Double, Int>> = both(doubleR, intR)
```

EXERCISE 6.7 (Hard)

If you can combine two RNG transitions, you should be able to combine a whole list of them. Implement `sequence` for combining a `List` of transitions into a single transition. Use it to reimplement the `ints` function you wrote before. For the sake of simplicity in this exercise, it is acceptable to write `ints` with recursion to build a list with `x` repeated `n` times.

```
fun <A> sequence(fs: List<Rand<A>>): Rand<List<A>> = TODO()
```

Once you're done implementing `sequence()`, try reimplementing it using a fold.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

6.4.2 Recursive retries through nested state actions

We've progressed from mutating existing state to passing state actions explicitly, after which we then developed a more elegant API to hide these transitions in the background. In doing so, we are beginning to see a pattern emerging: we're progressing toward implementations that don't explicitly mention or pass along the `RNG` value. The `map` and `map2` combinators allowed us to implement, in a rather succinct and elegant way, functions that were otherwise tedious and error-prone to write. But there are some functions that we can't very well write in terms of `map` and `map2`.

One such function is `nonNegativeLessThan`, which generates an integer between 0 (inclusive) and `n` (exclusive). A first stab at an implementation might be to generate a non-negative integer modulo `n`:

```
fun nonNegativeLessThan_A(n: Int): Rand<Int> =
    map(::nonNegativeInt) { it % n }
```

This will certainly generate a number in the range, but it'll be skewed because `Int.MaxValue` may not be exactly divisible by `n`. So numbers that are less than the remainder of that division will come up more frequently. When `nonNegativeInt` generates numbers higher than the largest multiple of `n` that fits in a 32-bit integer, we should *retry* the generator and hope to get a smaller number. We might attempt this:

Listing 6.10 Failed recursive retry with no state available.

```
fun nonNegativeLessThan_B(n: Int): Rand<Int> =
        map(::nonNegativeInt) { i ->
            val mod = i % n
            if (i + (n - 1) - mod >= 0) mod
            else nonNegativeLessThan_B(n)(???)   ❶   ❷
        }
```

❶ Retry recursively if the `Int` we got is higher than the largest multiple of `n` that fits in a 32-bit `Int`.

❷ Incorrect type of `nonNegativeLessThan(n)` fails compilation.

This is moving in the right direction, but `nonNegativeLessThan(n)` has the wrong type to be used right there. Remember, it should return a `Rand<Int>` which *is a function* that expects an `RNG`! But we don't have one right there. What we would like is to chain things together so that the `RNG` that's returned by `nonNegativeInt` is passed along to the recursive call to `nonNegativeLessThan`. We could pass it along explicitly instead of using `map`, like this:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 6.11 Successful recursive retry passing derived state explicitly.

```
fun nonNegativeIntLessThan(n: Int): Rand<Int> =
    { rng ->
        val (i, rng2) = nonNegativeInt(rng)
        val mod = i % n
        if (i + (n - 1) - mod >= 0)
            Pair(mod, rng2)
        else nonNegativeIntLessThan(n)(rng2)
    }
```

But it would be better to have a combinator that does this passing along for us. Neither `map` nor `map2` will cut it. We need a more powerful combinator, `flatMap`.

EXERCISE 6.8

Implement `flatMap`, and then use it to implement `nonNegativeLessThan`.

```
fun <A, B> flatMap(f: Rand<A>, g: (A) -> Rand<B>): Rand<B> = TODO()
```

`flatMap` allows us to generate a random `A` with `Rand<A>`, and then take that `A` and choose a `Rand<B>` based on its value. In `nonNegativeLessThan`, we use it to choose whether to retry or not, based on the value generated by `nonNegativeInt`.

EXERCISE 6.9

Reimplement `map` and `map2` in terms of `flatMap`. The fact that this is possible is what we're referring to when we say that `flatMap` is more *powerful* than `map` and `map2`.

6.4.3 Applying the combinator API to the initial example

In our quest to arrive at a more elegant approach handling state transitions, we have arrived at a clean API that employs combinators to seamlessly pass state in the background without any effort on our behalf.

We can now revisit our example from Section 6.1. Can we make a more testable die roll using our purely functional API?

Here's an implementation of `rollDie` using `nonNegativeLessThan`, including the off-by-one error we had before:

```
fun rollDie(): Rand<Int> =
    nonNegativeIntLessThan(6)
```

If we test this function with various `RNG` states, we'll pretty soon find an `RNG` that causes this function to return 0:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
>>> val zero = rollDie(SimpleRNG(5)).first
zero: Int = 0
```

And we can re-create this reliably by using the same `SimpleRNG(5)` random generator, without having to worry that its state is destroyed after being used.

Fixing the bug is trivial:

```
fun rollDie_B(): Rand<Int> =
    map(nonNegativeIntLessThan(6)) { it + 1 }
```

By using combinators we have greatly reduced complexity by no longer having to pass the random number generator through our program explicitly. We have defined a higher-level domain-specific language that greatly simplifies how we reason about a problem such as this simple off-by-one error.

6.5 A general state action data type

Even though we have only been working with state in random number generation, we can easily apply this technique to any other domain where passing state is required. On closer inspection we see that the combinators we have written aren't specific to any domain either, and can easily be utilized for passing *any* kind of state. In this section we intend to develop a data type that allows us to generalize over state transition of arbitrary kind.

As we've just discovered, the functions we've just written—`unit`, `map`, `map2`, `flatMap`, and `sequence`—aren't really specific to random number generation at all. They're general-purpose functions for working with state actions, and don't care about the type of the state. Note that, for instance, `map` doesn't care that it's dealing with `RNG` state actions, and so we can give it a more general signature replacing `RNG` with `S`:

Listing 6.12 Generalized version of `map` combinator

```
fun <S, A, B> map(
    sa: (S) -> Pair<A, S>,
    f: (A) -> B
): (S) -> Pair<B, S> = TODO()
```

Changing this signature doesn't require modifying the implementation of `map`! The more general signature was there all along; we just didn't see it.

We should then come up with a more general type than `Rand`, for handling any type of state:

Listing 6.13 Generalized `State` type alias for state transition

```
typealias State<S, A> = (S) -> Pair<A, S>
```

Here `State` is short for *computation that carries some state along*, or *state action*, *state*

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

transition, or even *statement* (see section 6.6). We might want to write it as its own class, wrapping the underlying function and naming it `run`.

Listing 6.14 Wrapping the state transition in a data class

```
data class State<S, out A>(val run: (S) -> Pair<A, S>)
```

With that said, the representation doesn't matter too much. What's important is that we have a single, general-purpose type, and using this type we can write general-purpose functions for capturing common patterns of stateful programs.

We can now just make `Rand` a type alias for `State`:

Listing 6.15 The `Rand` type alias updated to use `State`

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

EXERCISE 6.10

Generalize the functions `unit`, `map`, `map2`, `flatMap`, and `sequence`. Add them as methods on the `State` data class where possible. Alternatively, where it makes sense place them in the `State` companion object.

NOTE Always consider where a method should live—in the companion object, or on the class of the data type itself. In the case of a method that operates on an instance of the data type such as `map`, it certainly does make sense to place it at the class level. When we merely emit a value, such as is the case in the `unit` method, or if we operate on multiple instances such as in `map2` and `sequence`, it probably makes more sense to tie them to the companion object. This is often subject to individual taste and may differ depending on who is providing the implementation.

The functions we've written in Exercise 6.10 capture only a few of the most common patterns. As you write more functional code, you'll likely encounter other patterns and discover other functions to capture them.

6.6 Purely functional imperative programming

We begin to sacrifice readability as a result of the escalating complexity of our functional code. Is it possible to regain some of the readability we've lost in the process? Can we get back to something that resembles the simple imperative style that we know so well?

The good news is that we *can* achieve this simpler style of expression when we are coding in a pure functional way. In this section we will demonstrate how we can write functional code with

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

the *appearance* of it being imperative. We can achieve this using the *for-comprehension*, a concept that we briefly touched upon during chapter 4, but will now expand upon in far greater detail.

Up to now we have spent a lot of time developing our own implementation of `State`, and have learned a lot in doing so. We will now switch over to using an alternative implementation provided by Arrow. In Chapters 3 and 4 we already introduced Arrow as a functional companion for Kotlin. In particular, we learned about the `Option` and `Either` data types that are already provided, and their capability of working with for comprehensions. It so happens that the provided implementation of `State` also has this capability, which will enable the imperative style that we are after.

In the preceding sections, we were writing functions that followed a definite pattern. We'd run a state action, assign its result to a `val`, then run another state action that used that `val`, assign its result to another `val`, and so on. It looks a lot like *imperative* programming.

In the imperative programming paradigm, a program is a sequence of statements where each statement may modify the program state. That's exactly what we've been doing, except that our "statements" are really `State` actions, which are really functions. As functions, they read the current program state simply by receiving it in their argument, and they write to the program state simply by returning a value.

SIDEBAR **Are imperative and functional programming opposites?**

Absolutely not! Remember, functional programming is simply programming without side effects. Imperative programming is about programming with statements that modify some program state, and as we've seen, it's entirely reasonable to maintain state without side effects.

Functional programming has excellent support for writing imperative programs, with the added benefit that such programs can be reasoned about equationally because they're referentially transparent. We'll have much more to say about equational reasoning about programs in part 2, and imperative programs in particular in parts 3 and 4.

We implemented some combinators like `map`, `map2`, and ultimately `flatMap` to handle the propagation of the state from one statement to the next. But in doing so, we seem to have lost a bit of the imperative mood.

Consider as an example the following declarations. Here we care more about the type signatures than the implementations.

NOTE In the Arrow `State` class, the state type argument appears first.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 6.16 State declarations and combinators required to perform state propagation

```
val int: State<RNG, Int> = TODO()   ❶

fun ints(x: Int): State<RNG, List<Int>> = TODO()   ❷

fun <A, B> flatMap(
    s: State<RNG, A>,
    f: (A) -> State<RNG, B>
): State<RNG, B> = TODO()   ❸

fun <A, B> map(
    s: State<RNG, A>,
    f: (A) -> B
): State<RNG, B> = TODO()   ❹
```

❶ is a `State<RNG, Int>` with the ability of generating a single random integer

❷ returns a `State<RNG, List<Int>>` that can generate a list of `x` random integers

❸ a `flatMap` function that operates on a `State<RNG, A>` with a function from `A` to `State<RNG, B>`

❹ a `map` function that operates on a `State<RNG, A>` with a function that transforms `A` to `B`

Now we write some code utilizing these declarations that we subsequently have available to us from Listing 6.16.

Listing 6.17 State propagation using a series of `flatMap` and `map`s.

```
val ns: State<RNG, List<Int>> =
    flatMap(int) { x ->   ❶
        flatMap(int) { y ->   ❷
            map(ints(x)) { xs ->   ❸
                xs.map { it % y }   ❹
            }
        }
    }
```

❶ `int` will generate a single random integer.

❷ `ints(x)` generates a list of length `x` random integers

❸ replaces every element in the list with its remainder when divided by `y`

It's not clear what's going on here due to all the nested `flatMap` and `map` calls. Let's look for a simpler approach by turning our attention to the for-comprehension. This construct will unravel a series of `flatMap` calls, allowing us to rewrite this code in what *seems* to be a series of imperative declarations. The trick with this code lies in the destructuring that occurs in every step. Each time we see something being destructured, it implies a call to `flatMap`.

For example, a line that is written as `flatMap(int) { x -> ... }` could be rewritten as `val (x: Int) = int` within the confines of a for-comprehension. With this in mind, let's try rewriting Listing 6.17 using this technique.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 6.18 State propagation using a for-comprehension.

```
val ns2: State<RNG, List<Int>> =
    State.fx(Id.monad()) {   ❶
        val (x: Int) = int   ❷
        val (y: Int) = int   ❸
        val (xs: List<Int>) = ints(x)   ❹
        xs.map { it % y }   ❺
    }
```

❶ open the for-comprehension by passing a code block into `State.fx(Id.monad())`

❷ `int` is destructured to an `Int` named `x`

❸ `int` is again destructured to an `Int` named `y`

❹ `ints(x)` is destructured to a `List<Int>` of length `x`

❺ replace every element in `xs` with its remainder when divided by `y`, return the result

We begin by opening the for-comprehension using a call to `State.fx` and passing in an instance of `Id.monad()`. We won't concern ourselves very much with the detail regarding this, but suffice to say that this will allow us to pass in an anonymous function that acts like a block of imperative code and then return the final outcome of this block as a value.

The code block making up the for-comprehension is much easier to read (and write), and it looks like what it is—an imperative program that maintains some state. But it's the *same code* as in the previous example. We get the next `Int` and assign it to `x`, get the next `Int` after that and assign it to `y`, then generate a list of length `x`, and finally return the list with all of its elements modulo `y`.

We almost have everything we need in order to write fully fledged functional programs in an imperative style. To facilitate this kind of imperative programming with for-comprehensions (or `flatMaps`), we really only need two additional primitive `State` combinators—one for reading the state and one for writing the state. If we imagine that we have a combinator `get` for getting the current state, and a combinator `set` for setting a new state, we could implement a combinator that can modify the state in arbitrary ways:

Listing 6.19 Combinator to modify the current `State`

```
fun <S> modify(f: (S) -> S): State<S, Unit> =
    State.fx(Id.monad()) {   ❶
        val (s: S) = get<S>()   ❷
        val (_) = set(f(s))   ❸
    }
```

❶ Set up the for-comprehension for `State`

❷ Get the current state and assign it to `s`

❸ Set the new state of `f` applied to `s`

This method returns a `State` action that modifies the incoming state by the function `f`. It yields

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`Unit` to indicate that it doesn't have a return value other than the state.

What would the `get` and `set` actions look like? They're exceedingly simple. The `get` action simply passes the incoming state along and returns it as the value:

Listing 6.20 The `get` combinator retrieves, then passes on its state

```
fun <S> get(): State<S, S> =
    State { s -> Tuple2(s, s) }
```

The `set` action is constructed with a new state `s`. The resulting action ignores the incoming state, replaces it with the new state, and returns `Unit` instead of a meaningful value:

Listing 6.21 The `set` combinator updates the state, then returns `Unit`

```
fun <S> set(s: S): State<S, Unit> =
    State { Tuple2(s, Unit) }
```

The `get`, `set` and `modify` combinators can already be found on the `arrow.mtl.StateApi` class, but we just wanted to show what they entail for the purpose of demonstration. These two simple actions, together with the `State` combinators that we wrote—`unit`, `map`, `map2`, and `flatMap`—are all the tools we need to implement any kind of state machine or stateful program in a purely functional way.

TIP We recommend that you familiarize yourself with the Arrow documentation for `State` (arrow-kt.io/../-state.html), as well as to look at the underlying source code (github.com/../State.kt) in order fully grasp how it works. As mentioned before in section 1.5, the Arrow implementation is somewhat richer than the one that we've written, employing a monad transformer `StateT`, and exposing a public API `StateAPI` for the `State` class. Don't worry too much about what a monad transformer is, just be aware that the `StateApi` is responsible for exposing methods such as `get`, `set`, `modify` `stateSequential` and `stateTraverse` which *might* come in handy in the exercise that follows.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 6.11 (Hard/Optional)

To gain experience with the use of `State`, implement a finite state automaton that models a simple candy dispenser. The machine has two types of input: you can insert a coin, or you can turn the knob to dispense candy. It can be in one of two states: locked or unlocked. It also tracks how many candies are left and how many coins it contains.

```
sealed class Input

object Coin : Input()
object Turn : Input()

data class Machine(
    val locked: Boolean,
    val candies: Int,
    val coins: Int
)
```

The rules of the machine are as follows:

- Inserting a coin into a locked machine will cause it to unlock if there's any candy left.
- Turning the knob on an unlocked machine will cause it to dispense candy and become locked.
- Turning the knob on a locked machine or inserting a coin into an unlocked machine does nothing.
- A machine that's out of candy ignores all inputs.

The method `simulateMachine` should operate the machine based on the list of inputs and return the number of coins and candies left in the machine at the end. For example, if the input `Machine` has 10 coins and 5 candies, and a total of 4 candies are successfully bought, the output should be `(14, 1)`. Use the following declaration stubs to implement your solution:

```
fun simulateMachine(
    inputs: List<Input>
): State<Machine, Tuple2<Int, Int>> = TODO()
```

6.7 Conclusion

In this chapter, we touched on the subject of how to write purely functional programs that have state. We used random number generation as the motivating example, but the overall pattern comes up in many different domains. The idea is simple: use a pure function that accepts a state as its argument, and it returns the new state alongside its result. Next time you encounter an imperative API that relies on side effects, see if you can provide a purely functional version of it, and use some of the functions we wrote here to make working with it more convenient.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

6.8 Summary

- Updating state *explicitly* allows the recovery of referential transparency, lost in situations when mutating state.
- Stateful APIs are made pure by *computing* each subsequent state, not mutating existing state.
- A function that computes state based on a previous state is know as a *state action* or *state transition*.
- You can use *Combinators* to abstract over repetitive state transition patterns, and even to combine and nest state actions when required.
- You can adopt an *imperative* style when dealing with state transitions through the use of for-comprehensions.
- Arrow offers a useful `State` API and associated data type to model all the concepts dealt with in this chapter.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

7 Purely functional parallelism

This chapter covers:

- Designing a purely functional library through explorative design
- Choosing appropriate data types and functions to model the domain
- Reasoning about an API in terms of an algebra to discover types
- Defining laws to govern the behaviour of an API
- Generalizing combinators to make them broadly applicable
- Using pure functional parallelism as a working example of library design

Because modern computers have multiple cores per CPU, and often multiple CPUs, it's more important than ever to design programs in such a way that they can take advantage of this parallel processing power. But the interaction of parallel processes is complex, and the traditional mechanism for communication among execution threads—shared mutable memory—is notoriously difficult to reason about. This can all too easily result in programs that have race conditions and deadlocks, aren't readily testable, and don't scale well.

In this chapter, we'll build a purely functional library for creating parallel and asynchronous computations. We'll rein in the complexity inherent in parallel programs by describing them using only pure functions. This will let us use the substitution model to simplify our reasoning and hopefully make working with concurrent computations both easy and enjoyable.

What you should take away from this chapter is not only how to write a library for purely functional parallelism, but *how to approach the problem of designing a purely functional library*. Our main concern will be to make our library highly composable and modular. To this end, we'll keep with our theme of separating the concern of *describing* a computation from actually *running* it. We want to allow users of our library to write programs at a very high level, insulating them from the nitty-gritty of how their programs will be executed. For example,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

towards the end of the chapter we'll develop a combinator, `parMap`, that will let us easily apply a function `f` to every element in a collection simultaneously that looks something like this:

```
val outputList = parMap(inputList, f)
```

To get there, we'll work iteratively. We'll begin with a simple use case that we'd like our library to handle, and then develop an interface that facilitates this use case. Only then will we consider what our implementation of this interface should be. As we keep refining our design, we'll oscillate between the interface and implementation as we gain a better understanding of the domain and the design space through progressively more complex use cases. We'll introduce *algebraic reasoning* and demonstrate that an API can be described by *an algebra* that obeys specific *laws*.

Why design our own library? Why not just use the concurrency that comes with Kotlin's standard library through the use of coroutines? We'll design our own library for two reasons: the first is for pedagogical purposes to demonstrate how easy it is to design your own library. The second reason is that we want to encourage the view that no existing library is authoritative or beyond reexamination, even if designed by experts and labeled "standard." There's a certain safety in doing what everybody else does, but what's conventional isn't necessarily the most practical. Most libraries contain a lot of arbitrary design choices, many made unintentionally. When you start from scratch, you get to revisit all the fundamental assumptions that went into designing the library, take a different path, and discover things about the problem space that others may not have even considered. As a result, you might arrive at your own design that suits your purposes better. In this particular case, our fundamental assumption will be that our library permits *absolutely no side effects*.

We'll write a lot of code in this chapter, in part posed as exercises for you, the reader. As always, you can find hints and answers in appendix B and C at the back of the book.

7.1 Choosing data types and functions

When you begin designing a functional library, you usually have some general ideas about what you want to be able to do, and the difficulty in the design process is in refining these ideas and finding a data type that enables the functionality you want. In our case, we'd like to be able to "create parallel computations," but what does that mean exactly? Let's try to refine this into something we can implement by examining a simple, parallelizable computation—summing a list of integers. The usual left fold for this would be as follows:

```
fun sum(ints: List<Int>): Int =
    ints.foldLeft(0) { a, b -> a + b }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

NOTE For ease of use we aren't using our own `List` implementation from chapter 3, but have opted for the `List` provided by the Kotlin standard library. This list implementation exposes a read-only interface, although the underlying implementation is mutable. We are willing to make this compromise because we won't be using any advanced list features, and as library authors want to keep the dependency graph as small as possible.

Instead of folding sequentially, we could use a divide-and-conquer algorithm like in listing 7.1.

Listing 7.1 Summing a list using a divide-and-conquer approach

```
fun sum(ints: List<Int>): Int =
    if (ints.size <= 1)
        ints.firstOption().getOrElse { 0 }   ❶
    else {
        val (l, r) = ints.splitAt(ints.size / 2)   ❷
        sum(l) + sum(r)   ❸
    }
```

❶ Deal with cases of 1 or 0 ints, using Arrow extension method `firstOption`, like `headOption` in chapter 3.

❷ Split and destructure the list into two using helper extension method `splitAt`

❸ Recursive call to `sum` for both `l` and `r` and sum them up

We divide the sequence in half using the `splitAt` function, recursively sum both halves, and then combine their results. And unlike the `foldLeft`-based implementation, this implementation *can* be parallelized with the two halves being summed in parallel.

SIDEBAR **The importance of simple examples**

Summing integers in practice is so fast that parallelization imposes more overhead than it saves. But simple examples like this are exactly the kind that are most helpful to consider when designing a functional library.

Complicated examples include all sorts of incidental detail that can confuse the initial design process. We're trying to explain the essence of the problem domain, and a good way to do that is to start with trivial examples, factor out common concerns across these examples, and gradually add complexity.

In functional design, our goal is to achieve expressiveness without numerous special cases, rather building a simple and composable set of core data types and functions.

As we think about what sort of data types and functions could enable parallelizing this computation, we begin to shift our perspective. Rather than focusing on how this parallelism will be implemented and forcing ourselves to work with the underlying APIs directly, we turn the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

tables. Instead we'll design our own shiny new API as illuminated by our examples, working backward from there to our own implementation that uses underlying libraries such as `java.concurrent` to do the heavy lifting.

7.1.1 A data type for parallel computations

Let's take a closer look at the line `sum(l) + sum(r)` in Listing 7.1, which invokes `sum` on the two halves recursively. We immediately see that any data type we might choose to represent our parallel computation needs to be able to *contain a result*. This result will have some meaningful type (in this case `Int`). We also require some way of extracting this result. Let's apply this newfound knowledge to our design. For now, we can just invent a container type for our result, call it `Par<A>` (for *parallel*) and legislate the existence of the functions we need:

Listing 7.2 Defining a new data type for parallelism

```
class Par<A>(val get: A)   ❶

fun <A> unit(a: () -> A): Par<A> = Par(a())   ❷

fun <A> get(a: Par<A>): A = a.get   ❸
```

❶ A new data type to contain a result

❷ Create a unit of parallelism from unevaluated `A`

❸ Extract the evaluated result of `A`

Can we really do such a thing? Yes, of course! For now, we don't need to worry about what other functions we require, what the internal representation of `Par` might be, or how these functions are implemented. We are simply conjuring up the needed data type and its associated functions to meet the needs of our simple example. Let's revisit our example from Listing 7.1 now.

Listing 7.3 Using our new data type to assimilate parallelism

```
fun sum(ints: List<Int>): Int =
    if (ints.size <= 1)
        ints.firstOption().getOrElse { 0 }
    else {
        val (l, r) = ints.splitAt(ints.size / 2)
        val sumL: Par<Int> = unit { sum(l) }   ❶
        val sumR: Par<Int> = unit { sum(r) }   ❷
        sumL.get + sumR.get   ❸
    }
```

❶ Compute left side of list in context of `Par`

❷ Compute right side of list in context of `Par`

❸ Extract `Int` results from either `Par` and sum them

We have now added our new `Par` data type to the mix. We wrap both recursive calls to `sum` in `Par` using the `unit` factory method, which in turn is responsible for evaluating all calls to `sum`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Next we extract both results from their `Par`s in order to sum them up.

We now have a choice about the meaning of `unit` and `get`—`unit` could begin evaluating its argument immediately in a separate logical thread,[34] or it could simply defer evaluation of its argument until `get` is called. But note that in Listing 7.3, if we want to obtain any degree of parallelism we require that `unit` begin evaluating its argument concurrently and immediately return without blocking. Can you see why? Function arguments in Kotlin are strictly evaluated from left to right, so if `unit` delays execution until `get` is called, we will spawn the parallel computation *and* wait for it to finish before spawning the second parallel computation. This means the computation is effectively sequential!

But if `unit` begins evaluating its argument concurrently, then calling `get` is responsible for breaking referential transparency. We can see this by replacing `sumL` and `sumR` with their definitions—if we do so, we still get the same result, but our program is no longer parallel, as can be seen here:

```
unit { l }.get + unit { r }.get
```

If `unit` starts evaluating its argument right away, the next thing to happen is that `get` will wait for that evaluation to complete. So the two sides of the `+` sign won't run in parallel if we simply inline the `sumL` and `sumR` variables. We can see that `unit` has a definite side effect, but *only* in conjunction with `get`. We say this because `unit`, which merely represents an asynchronous computation `Par<Int>`, will block execution when we call `get`, this in turn exposing the side effect. So we should avoid calling `get`, or at least delay calling it until the very end. We seek to combine asynchronous computations without waiting for them to complete.

Before we continue, let's reflect on what we've done. First, we conjured up a simple, almost trivial example. Next, we explored this example to uncover a design choice. Then, via some experimentation, we discovered an interesting consequence of one option, and in the process learned something fundamental about the nature of our problem domain! The overall design process is a series of small adventures. You don't need a special license to do such exploration, and you certainly don't need to be an expert in functional programming either. Just dive in and see what you can find!

7.1.2 Combining parallel computations to ensure concurrency

Let's see if we can avoid the aforementioned pitfall of combining `unit` and `get`. If we don't call `get`, that implies that our `sum` function must return a `Par<Int>`. What consequences does this change reveal? Again, let's just invent a function, say `map2`, with the required signature:

```
fun sum(ints: List<Int>): Par<Int> =
    if (ints.size <= 1)
        unit { ints.firstOption().getOrElse { 0 } }
    else {
        val (l, r) = ints.splitAt(ints.size / 2)
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        map2(sum(l), sum(r)) { lx: Int, rx: Int -> lx + rx }
    }
```

EXERCISE 7.1

The higher-order function `map2` is a new function for combining the result of two parallel computations. What is its signature? Give the most general signature possible (don't assume it works only for `Int`).

Observe that we're no longer calling `unit` in the recursive case, and it isn't clear whether `unit` should accept its argument lazily anymore. In this example, accepting the argument lazily doesn't seem to provide any benefit, but perhaps this isn't always the case. Let's come back to this question later.

What about `map2`—should it take its arguments lazily? Would it make sense for `map2` to run both sides of the computation in parallel giving each side equal opportunity to run? It would seem arbitrary for the order of the `map2` arguments to matter as we simply want it to indicate that the two computations being combined are independent and can be run in parallel. What choice lets us implement this meaning? As a simple example, consider what happens if `map2` is strict in both arguments as we evaulate `sum(listOf(1, 2, 3, 4))`. Take a minute to work through and understand the following (somewhat stylized) program trace.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 7.4 Strict evaluation of both parameters results in left side being evaluated first

```
sum(listOf(1, 2, 3, 4))   ❶

map2(
    sum(listOf(1, 2)),
    sum(listOf(3, 4))
) { i: Int, j: Int -> i + j }   ❶

map2(
    map2(
        sum(listOf(1)),
        sum(listOf(2))
    ) { i: Int, j: Int -> i + j },   ❸
    sum(listOf(3, 4))
) { i: Int, j: Int -> i + j }

map2(
    map2(
        unit { 1 },
        unit { 2 }
    ) { i: Int, j: Int -> i + j },   ❹
    sum(listOf(3, 4))
) { i: Int, j: Int -> i + j }

map2(
    map2(
        unit { 1 },
        unit { 2 }
    ) { i: Int, j: Int -> i + j },
    map2(
        sum(listOf(3)),
        sum(listOf(4))
    ) { i: Int, j: Int -> i + j }   ❺
) { i: Int, j: Int -> i + j }
```

❶ Unevaluated expression

❷ Substitute `sum` with its definition of `map2`

❸ Substitute *left* argument with its definition `map2`

❹ Substitute left `sum` expressions with their results

❺ Substitution of *right* hand side with its definition

To evaluate `sum(x)` in Listing 7.4, we substitute `x` into the definition of `sum`, as we've done in previous chapters. Because `map2` is strict, and Kotlin evaluates arguments left to right, whenever we encounter `map2(sum(x),sum(y)) { i, j -> i + j }`, we then have to evaluate `sum(x)` and so on recursively. This has the unfortunate consequence that we'll strictly construct the entire left half of the tree of summations first before moving on to (strictly) constructing the right half. Here `sum(listOf(1,2))` gets fully expanded before we consider `sum(listOf(3,4))`. And if `map2` evaluates its arguments in parallel (using whatever resource is being used to implement the parallelism, like a thread pool), that implies the left half of our computation will start executing before we even begin constructing the right half of our computation.

What if we keep `map2` strict, but *don't* let it begin execution immediately? Does this help? If

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`map2` doesn't begin evaluation immediately, this implies a `Par` value is merely constructing a *description* of what needs to be computed in parallel. Nothing actually occurs until we *evaluate* this description, perhaps by using a `get`-like function. The problem is that if we construct our descriptions strictly, they'll be rather heavyweight objects. Looking at the trace in Listing 7.5, our description will have to contain the full tree of operations to be performed.

Listing 7.5 Strict construction of the description results in a full tree of operations

```
map2(
    map2(
        unit { 1 },
        unit { 2 }) { i: Int, j: Int -> i + j },
    map2(
        unit { 3 },
        unit { 4 }) { i: Int, j: Int -> i + j }
) { i: Int, j: Int -> i + j }
```

Whatever data structure we use to store this description, it'll likely occupy more space than the original list itself! It would be nice if our descriptions were more lightweight. It also seems that we should make `map2` lazy and have it begin immediate execution of both sides in parallel. This also addresses the problem of giving either side priority over the other.

7.1.3 Marking computations to be forked explicitly

Something still doesn't feel right about our latest choice. Is it *always* the case that we want to evaluate the two arguments to `map2` in parallel? Probably not. Consider this simple hypothetical example:

```
map2(
    unit { 1 },
    unit { 2 }
) { i: Int, j: Int -> i + j }
```

In this case, we happen to know that the two computations we're combining will execute so quickly that there isn't any point in spawning a new logical thread to evaluate them. But our API doesn't give us any way of providing this sort of information. That is, our current API is very *inexplicit* about when computations get forked off the main thread into a new thread process—the programmer doesn't get to specify *where* this forking should occur. What if we make the forking more explicit? We can do that by inventing another function which we can take to mean that the given `Par` should be run in a separate logical thread:

```
fun <A> fork(a: () -> Par<A>): Par<A> = TODO()
```

Applying this function to our running `sum` example, we could express it as follows:

```
fun sum(ints: List<Int>): Par<Int> =
    if (ints.size <= 1)
        unit { ints.firstOption().getOrElse { 0 } }
    else {
        val (l, r) = ints.splitAt(ints.size / 2)
        map2(
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
            fork { sum(l) },
            fork { sum(r) }
        ) { lx: Int, rx: Int -> lx + rx }
    }
```

With `fork`, we can now make `map2` strict, leaving it up to the programmer to wrap arguments if they wish. A function like `fork` solves the problem of instantiating our parallel computations too strictly, but more fundamentally it puts the parallelism explicitly under programmer control. We're addressing two concerns here. The first is that we need some way to indicate that the results of the two parallel tasks should be combined. Apart from this, we have the choice of whether a particular task should be performed asynchronously. By keeping these concerns separate, we avoid having any sort of global policy for parallelism attached to `map2` and other combinators we write, which would mean making tough and ultimately arbitrary choices about what global policy is best.

Let's now return to the question of whether `unit` should be strict or lazy. With `fork`, we can now make `unit` strict without any loss of expressiveness. A non-strict version of it, let's call it `lazyUnit`, can be implemented using `unit` and `fork`.

Listing 7.6 A strict `unit` and `fork` can be combined to form a lazy variant of `unit`.

```
fun <A> unit(a: A): Par<A> = Par(a)

fun <A> lazyUnit(a: () -> A): Par<A> =
    fork { unit(a()) }
```

The function `lazyUnit` is a simple example of a *derived* combinator, as opposed to a *primitive* combinator like `unit`. We were able to define `lazyUnit` in terms of other operations only. Later, when we pick a representation for `Par`, `lazyUnit` won't need to know anything about this representation—its only knowledge of `Par` is through the operations `fork` and `unit` that are defined on `Par`.

We know we want `fork` to signal that its argument gets evaluated in a separate logical thread. But we still have the question of whether it should begin doing so *immediately* upon being called, or else hold on to its argument to be evaluated in a logical thread later when the computation is *forced* using something like `get`. In other words, should evaluation be the responsibility of `fork` or of `get`? Should evaluation be eager or lazy? When you're unsure about a meaning to assign to some function in your API, you can always continue with the design process—at some point later the trade-offs of different choices of meaning may become clear. Here we make use of a helpful trick—we'll think about what sort of *information* is required to implement `fork` and `get` with various meanings.

If `fork` begins evaluating its argument immediately in parallel, the implementation must clearly know something, either directly or indirectly, about how to create threads or submit tasks to some sort of thread pool. This implies that the thread pool or whatever resource we use to

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

implement the parallelism must be globally accessible and properly initialized wherever we want to call `fork`. This means we lose the ability to control the parallelism strategy used for different parts of our program. And though there's nothing wrong with having a global resource for executing parallel tasks, we can imagine how it would be useful to have more fine-grained control over what implementations are used and in what context. For instance, we might like each subsystem of a large application to get its own thread pool with different parameters. As we consider this, it seems much more appropriate to give `get`, the responsibility of creating threads and submitting execution tasks.

Note that coming to these conclusions didn't require knowing exactly how `fork` and `get` would be implemented, nor even what the representation of `Par` would be. We just reasoned informally about the sort of information required to actually spawn a parallel task, then examined the consequences of having `Par` values know about this information.

If `fork` simply holds on to its unevaluated argument until later, it requires no access to the mechanism for implementing parallelism. It just takes an unevaluated `Par` and "marks" it for concurrent evaluation. With this model, `Par` itself doesn't need to know how to actually *implement* the parallelism. It's more a *description* of a parallel computation that gets *interpreted* at a later time by something like the `get` function. This is a shift from before, where we were considering `Par` to be a *container* of a value that we could simply *get* when it becomes available. Now it's more of a first-class *program* that we can *run*. Keeping this new discovery in mind, let's rename our `get` function to `run`, and dictate that this is where the parallelism actually gets implemented.

```
fun <A> run(a: Par<A>): A = TODO()
```

Because `Par` is now just a pure data structure, `run` has to have some means of implementing the parallelism, whether it spawns new threads, delegates tasks to a thread pool, or uses some other mechanism.

7.2 Picking a representation

Just by exploring this simple example and thinking through the consequences of different choices, we've arrived at the following API.

```
fun <A> unit(a: A): Par<A>   ❶

fun <A, B, C> map2(
    a: Par<A>,
    b: Par<B>,
    f: (A, B) -> C
): Par<C>   ❶

fun <A> fork(a: () -> Par<A>): Par<A>   ❷

fun <A> lazyUnit(a: () -> A): Par<A>   ❸

fun <A> run(a: Par<A>): A   ❺
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

1. Create a computation that immediately results in the value a
2. Combine the results of two parallel computations with a binary function.
3. Mark a computation for concurrent evaluation by `run`.
4. Wrap expression a for concurrent evaluation by `run`.
5. Fully evaluate a given `Par`, spawning computations and extracting value.

EXERCISE 7.2

At any point while evolving an API, you can start thinking about possible *representations* for the abstract types that appear. Try to come up with a representation for `Par` that makes it possible to implement the functions of our API.

Let's see if we can come up with a representation together. We know that `run` somehow needs to execute asynchronous tasks. We could write our own low-level API, but there's already a class that we can use in the Java Standard Library, `java.util.concurrent.ExecutorService`. Here is what the API looks like, roughly paraphrased in Kotlin.

Listing 7.7 Executor API represented in Kotlin.

```
interface Callable<A> {
    fun call(): A
}

interface Future<A> {
    fun get(): A
    fun get(timeout: Long, timeUnit: TimeUnit): A
    fun cancel(evenIfRunning: Boolean): Boolean
    fun isDone(): Boolean
    fun isCancelled(): Boolean
}

interface ExecutorService {
    fun <A> submit(c: Callable<A>): Future<A>
}
```

The `ExecutorService` allows us to submit a `Callable`, this being the equivalent to a lazy argument, or thunk, in Kotlin. The result from calling `submit` will be a `Future`, which is a handle to a computation that is potentially running in a new thread. We can query the `Future` for a result by calling one of its blocking `get` methods. It also sports some additional methods for canceling and querying its current state.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **The problem with using concurrency primitives directly**

What of `java.lang.Thread` and `Runnable`? Let's take a look at these classes. Here's a partial excerpt of their API, paraphrased in Kotlin:

```
interface Runnable {
    fun run(): Unit
}

class Thread(r: Runnable) {
    fun start(): Unit = TODO()   ❶
    fun join(): Unit = TODO()   ❷
}
```

❶ Begins running `r` in a separate thread.

❷ Blocks the calling thread until `r` finishes running.

Already, we can see a problem with both of these types—none of the methods return a meaningful value. Therefore, if we want to get any information out of a `Runnable`, it has to have some side effect, like mutating some state that we can inspect. This is bad for compositionality—we can't manipulate `Runnable` objects generically since we always need to know something about their internal behavior. `Thread` also has the disadvantage that it maps directly onto operating system threads, which are a scarce resource. It would be preferable to create as many "logical threads" as is natural for our problem, and later deal with mapping these onto actual OS threads.

You can handle this kind of thing by using something like `java.util.concurrent.Future`, `ExecutorService` and friends. Why don't we use them directly? Here's a paraphrased portion of their API:

```
class ExecutorService {
    fun <A> submit(a: Callable<A>): Future<A> = TODO()
}

interface Future<A> {
    fun get(): A
}
```

Though `java.util.concurrent` is a tremendous help in abstracting over physical threads, these primitives are still at a much lower level of abstraction than the library we want to create in this chapter. A call to `Future.get`, for example, blocks the calling thread until the `ExecutorService` has finished executing it, and its API provides no means of *composing* futures. Of course, we can build the implementation of our library on top of these tools (and this is what we end up doing later in the chapter), but they don't present the modular and compositional API that we'd want to use directly from functional programs.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Now try to imagine how we could modify `run` in our `Par` data type if we had access to an instance of the `ExecutorService`.

```
fun <A> run(es: ExecutorService, a: Par<A>): A = TODO()
```

The simplest possible way of expressing `Par<A>` might be to turn it into a type alias of a function such as `(ExecutorService) -> A`. If we invoke this function with an instance of an `ExecutorService`, it produces something of type `A`, making the implementation trivial. We could improve this further by giving the caller of `run` the ability to defer the decision of how long to wait for a computation, or even to cancel it all together. With this in mind, `Par<A>` becomes `(ExecutorService) -> Future<A>` with `run` simply returning the `Future<A>`.

```
typealias Par<A> = (ExecutorService) -> Future<A>

fun <A> run(es: ExecutorService, a: Par<A>): Future<A> = a(es)
```

Note that since `Par` is now represented by a *function* that needs an `ExecutorService`, the creation of the `Future` doesn't actually happen until this `ExectorService` is provided.

Is it really that simple? Let's assume it is for now, and revise our model if we find that it doesn't fulfill our requirements in the future.

7.3 Refining the API with the end user in mind

The way we've worked so far is a bit artificial. In practice, there aren't such clear cut boundaries between designing the API and choosing a representation with one preceding the other. Ideas for a representation can drive the API design, but the opposite may also happen—the choice of API can drive the representation. It is natural to shift fluidly between these two perspectives, run experiments when questions arise, build prototypes, and so on.

We'll devote this section to exploring and refining our API. Though we've already obtained a lot of mileage out of evolving this simple example, let's try to learn more about what we can express using the primitive operations that we already built before adding any new ones. With our primitives and their chosen meanings, we've carved out a small universe for ourselves. We now get to discover what ideas can be expressed in this universe. This can, and should be a fluid process—we can change the rules of our universe at any time, make a fundamental change to our representation or introduce a new primitive, all while observing how our creation subsequently behaves.

We will begin by implementing the functions of the API that we've developed up to this point. Now that we have a representation for `Par`, a first attempt should be straightforward. What follows is a naive implementation using this initial representation.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 7.8 Primitive operations for `Par` can be found in the `Pars` object

```
object Pars {
    fun <A> unit(a: A): Par<A> =
        { es: ExecutorService -> UnitFuture(a) }   ❶

    data class UnitFuture<A>(val a: A) : Future<A> {

        override fun get(): A = a

        override fun get(timeout: Long, timeUnit: TimeUnit): A = a

        override fun cancel(evenIfRunning: Boolean): Boolean = false

        override fun isDone(): Boolean = true

        override fun isCancelled(): Boolean = false
    }

    fun <A, B, C> map2(
        a: Par<A>,
        b: Par<B>,
        f: (A, B) -> C
    ): Par<C> =   ❷
        { es: ExecutorService ->
            val af: Future<A> = a(es)
            val bf: Future<B> = b(es)
            UnitFuture(f(af.get(), bf.get()))   ❸
        }

    fun <A> fork(
        a: () -> Par<A>
    ): Par<A> =   ❹
        { es: ExecutorService ->
            es.submit(Callable<A> { a()(es).get() })
        }
}
```

❶ `unit` represented as a function that returns a `UnitFuture`

❷ `map2` responsible only for combinatorial logic, no implicit threading

❸ Timeouts are not respected due to the calls to `get()`

❹ `fork` is not truly running in parallel due to blocking call `a()`

As stated before, this example is a naive solution to the problem. We will now identify the issues and address them one by one. The `unit` operator is represented as a function that returns a `UnitFuture`, which is a simple implementation of `Future` that simply wraps a constant value and never uses the `ExecutorService`. It is always executed and can't be canceled. Its `get` method simply returns the value that we gave it.

Next up is the `map2` operator. It doesn't evaluate the call to `f` in a separate logical thread, in accordance with our design choice of having `fork` be the sole function in the API that controls parallelism. We could always wrap `map2` with a call to `fork` if we wanted the evaluation of `f` to occur in a separate thread.

This implementation of `map2` also does not respect timeouts. It simply passes the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`ExecutorService` on to both `Par` values, waits for the results of the `Futures` `af` and `bf`, applies `f` to them, and finally wraps them in a `UnitFuture`. In order to respect timeouts, we'd need a new `Future` implementation that records the amount of time spent evaluating `af`, and then subtracts that time from the available time allocated for evaluating `bf`.

The `fork` operator has the simplest and most natural implementation possible, but there are also some problems here—for one, the outer `Callable` will block waiting for the "inner" task to complete. Since this blocking occupies a thread or resource backing our `ExecutorService`, we are losing out on some potential parallelism. This is a symptom of a more serious problem with the implementation that we'll discuss later in the chapter.

We should note that `Future` doesn't have a purely functional interface. This is part of the reason why we don't want users of our library to deal with `Future` directly. An important point to make is that even though methods on `Future` rely on side effects, our entire `Par` API remains pure. It's only after the user calls `run` and the implementation receives an `ExecutorService` that we expose the `Future`'s machinery. Our users are therefore programming to a pure interface with an implementation that relies on effects. But since our API remains pure, these effects aren't *side* effects. In part 4 we'll discuss this distinction in detail.

EXERCISE 7.3 (Hard)

Fix the implementation of `map2` so that it respects the contract of timeouts on `Future`.

EXERCISE 7.4

This API already enables a rich set of operations. As an example, using `lazyUnit` write a function to convert any function `(A) -> B` to one that evaluates its result asynchronously.

```
fun <A, B> asyncF(f: (A) -> B): (A) -> Par<B> = TODO()
```

What else can we express with our existing combinators? Let's look at a more concrete example.

Suppose we have a `Par<List<Int>>` representing a parallel computation that produces a `List<Int>`, and we'd like to convert this to a `Par<List<Int>>` with a sorted result:

```
fun sortPar(parList: Par<List<Int>>): Par<List<Int>> = TODO()
```

We could of course `run` the `Par`, sort the resulting list, and repackage it in a `Par` with `unit`. But we want to avoid calling `run`. The only other combinator we have that allows us to manipulate

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

the value of a `Par` in any way is `map2`. So if we passed `parList` to one side of `map2`, we'd be able to gain access to the `List` inside and sort it. And we can pass whatever we want to the other side of `map2`, so let's just pass a `Unit`:

```
fun sortPar(parList: Par<List<Int>>): Par<List<Int>> =
    map2(parList, unit(Unit)) { a, _ -> a.sorted() }
```

That was easy. We can now tell a `Par<List<Int>>` that we'd like that list sorted. But we might as well generalize this further. We can "lift" any function of type `(A) -> B` to become a function that takes `Par<A>` and returns `Par<B>`; we can `map` any function over a `Par`:

```
fun <A, B> map(pa: Par<A>, f: (A) -> B): Par<B> =
    map2(pa, unit(Unit), { a, _ -> f(a) })
```

As a result, `sortPar` now simply becomes:

```
fun sortPar(parList: Par<List<Int>>): Par<List<Int>> =
    map(parList) { it.sorted() }
```

That's terse and clear. We just combined the operations to make the types line up. And yet, if you look at the implementations of `map2` and `unit`, it should be clear this implementation of `map` *means* something sensible.

Was it cheating to pass the bogus value `unit(Unit)` as an argument to `map2`, only to ignore its value? Not at all! The fact that we can implement `map` in terms of `map2`, but not the other way around shows that `map2` is strictly more powerful than `map`. This sort of thing happens a lot when we're designing libraries—often, a function that seems to be primitive will turn out to be expressible using some more powerful primitive.

What else can we implement using our API? Could we `map` over a list in parallel? Unlike `map2`, which combines two parallel computations, `parMap` (let's call it) needs to combine *N* parallel computations. It seems like this should somehow be expressible:

```
fun <A, B> parMap(
    ps: List<A>,
    f: (A) -> B
): Par<List<B>> = TODO()
```

We could always just write `parMap` as a new primitive. Remember that `Par<A>` is simply a type alias for `(ExecutorService) -> Future<A>`.

There's nothing wrong with implementing operations as new primitives. In some cases we can even implement the operations more efficiently by assuming something about the underlying representation of the data types we're working with. But right now we're interested in exploring what operations are expressible using our existing API, and grasping the relationships between

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

the various operations we've defined. Understanding what combinators are truly primitive will become more important in part 3 when we show how to abstract over common patterns across libraries.

There is also another good reason not to implement `parMap` as a new primitive—it is challenging to do correctly, particularly if we want to respect timeouts properly. It is frequently the case that primitive combinators encapsulate some rather tricky logic, and reusing them means we don't have to duplicate this logic.

Let's see how far we can get implementing `parMap` in terms of existing combinators:

```
fun <A, B> parMap(
    ps: List<A>,
    f: (A) -> B
): Par<List<B>> {
    val fbs: List<Par<B>> = ps.map(asyncF(f))
    TODO()
}
```

Remember, `asyncF` converts an `(A) -> B` to an `(A) -> Par<B>` by forking a parallel computation to produce the result. So we can fork off our *N* parallel computations pretty easily, but we need some way of collecting their results. Are we stuck? Well, just from inspecting the types we can see that we need some way of converting our `List<Par<B>>` to the `Par<List<B>>` required by the return type of `parMap`.

EXERCISE 7.5 (Hard)

Write this function, called `sequence`. No additional primitives are required. Do not call `run`.

```
fun <A> sequence(ps: List<Par<A>>): Par<List<A>> = TODO()
```

Once we have `sequence`, we can complete our implementation of `parMap`:

```
fun <A, B> parMap(
    ps: List<A>,
    f: (A) -> B
): Par<List<B>> = fork {
    val fbs: List<Par<B>> = ps.map(asyncF(f))
    sequence(fbs)
}
```

Note that we've wrapped our implementation in a call to `fork`. With this implementation, `parMap` will return immediately, even for a huge input list. When we later call `run`, it will fork a single asynchronous computation which itself spawns *N* parallel computations, and then waits for these computations to finish, collecting their results into a list.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 7.6

Implement `parFilter`, which filters elements of a list in parallel.

```
fun <A> parFilter(
    sa: List<A>,
    f: (A) -> Boolean
): Par<List<A>> = TODO()
```

Can you think of any other useful functions to write? Experiment with writing a few parallel computations of your own to see which ones can be expressed without additional primitives. Here are some ideas to try:

- Is there a more general version of the parallel summation function we wrote at the beginning of this chapter? Try using it to find the maximum value of a `List` in parallel.
- Write a function that takes a list of paragraphs (a `List<String>`) and returns the total number of words across all paragraphs, in parallel. Generalize this function as much as possible.
- Implement `map3`, `map4`, and `map5`, in terms of `map2`.

7.4 Reasoning about the API in terms of algebraic equations

As the previous section demonstrates, we often get far by simply writing down the type signature for an operation we want, and then "following the types" to an implementation. When working this way, we can almost forget the concrete domain (for instance, when we implemented `map` in terms of `map2` and `unit`) and just focus on lining up types. This isn't cheating; it's a natural style of reasoning analogous to what we do when simplifying an algebraic equation. We're treating the API as an *algebra*,[35] or an abstract set of operations along with a set of *laws* or properties we assume to be true, and simply doing formal symbol manipulation following the rules of the game specified by this algebra.

Up until now, we've taken an informal approach to reasoning about our API. There's nothing wrong with this, but let's take a step back and formalize some laws we would like our API to hold. Without realizing it, we've mentally built up a model of what properties or laws we expect. By articulating them we can highlight design choices that wouldn't otherwise be apparent when reasoning informally. Two laws that come to mind are the laws of *mapping* and *forking* that we will discuss next in this section.

7.4.1 The law of mapping

Like any design choice, choosing laws has profound consequences—it places constraints on what the operations mean, determines the possible implementation choices, and affects what other properties can be true. Let's look at an example where we'll make up some law that seems feasible. This might be used as a test case if we were writing tests for our library.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
map(unit(1)) { it + 1 } == unit(2)
```

We're saying that mapping over `unit(1)` with the `{ it + 1 }` function is in some sense equivalent to `unit(2)`. Laws often start out this way, as concrete examples of *identities* we expect to hold. Here we mean identity in the mathematical sense of a statement that two expressions are identical or equivalent. In what sense are they equivalent? This is an interesting question, in light of the fact that `Par` is a simple function of `(ExecutorService) -> Future`. For now, let's say two `Par` objects are equivalent if *for any valid* `ExecutorService` argument, their `Future` results have the same value.

Laws and functions share much in common. Just as we can generalize functions, we can generalize laws. For instance, the preceding expression could be generalized in this way:

```
map(unit(x), f) == unit(f(x))
```

Here we're saying this should hold true for *any* choice of `x` and `f`, not just `1` and the `{ it + 1 }` function. This places some constraints on our implementation. Our implementation of `unit` can't inspect the value it receives and decide to return a parallel computation with a result of `42` when the input is `1`—it can only pass along whatever it receives. Similarly for our `ExecutorService`—when we submit `Callable` objects to it for execution, it can't make any assumptions or change behavior based on the values it receives. More concretely, this law disallows downcasting or `is` checks (often grouped under the term *typecasting*) in the implementations of `map` and `unit`.

Much like we strive to define functions in terms of simpler functions, each doing one thing only, we can also define laws in terms of simpler laws that each affirm one thing. Let's see if we can simplify this law further. We said we wanted this law to hold for *any* choice of `x` and `f`. Something interesting happens if we substitute the *identity function* for `f`. An identity function simply passes along its value, and can be defined as `fun <A> id(a: A): A = a`. We can now simplify both sides of the equation and get a new law that's considerably simpler, much like the same sort of substitution and simplification one might do when solving an algebraic equation.

Listing 7.9 Using the substitution model to simplify both sides of an equation

```
val x = 1
val y = unit(x)
val f = { a: Int -> a + 1 }
val id = { a: Int -> a }

map(unit(x), f) == unit(f(x))   ❶
map(unit(x), id) == unit(id(x))   ❷
map(unit(x), id) == unit(x)   ❸
map(y, id) == y   ❹
```

❶ Initial law declared

❷ Substitute `f` with identity function `id`

❸ Simplify `id(x)` to `x`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❹ Substitute `unit(x)` with its equivalent `y`

This is fascinating! Our simplified law talks about `map` only, leaving the mention of `unit` as an extraneous detail. To get some insight into what this new law suggests, let's think about what `map` *can't* do. It can't throw an exception and crash the computation before applying the function to the result. Can you see why this violates the law? All it *can* do is apply the function `f` to the result of `y`, which in turn leaves `y` unaffected when that function is `id`. We say that `map` is required to be *structure-preserving* in that it doesn't alter the structure of the parallel computation, only the value "inside" the computation.

7.4.2 The law of forking

This particular law doesn't do much to constrain our implementation of `Par`. You've probably been assuming these properties without even realizing it. It would be strange to have any special cases in the implementations of `map`, `unit`, `ExecutorService.submit` or have `map` randomly throwing exceptions. Let's consider a stronger property—that `fork` should not affect the result of a parallel computation:

```
fork { x } == x
```

This declaration seems obvious and *should* be true of our implementation. It is clearly a desirable property that is consistent with our expectation of how `fork` should work. `fork(x)` should do the same thing as `x`, albeit asynchronously in a logical thread separate from the main thread. If this law didn't always hold true, we'd have to know when it was safe to call without changing its meaning and without any help from the type system.

Surprisingly, this simple property places strong constraints on our implementation of `fork`. After you've written down a law like this, take off your implementer hat, put on your testing hat, and try to break your law. Think through any possible corner cases, try to come up with counterexamples, and even construct an informal proof that the law holds—at least enough to convince a skeptical fellow programmer.

BREAKING THE LAW: A SUBTLE BUG

Let's try this mode of thinking: we're expecting that `fork(x) == x` for *all* choices of `x`, and any choice of `ExecutorService`. We have a good sense of what `x` could be—it's some expression making use of `fork`, `unit`, `map2`, or any other possible combinators derived from these. What about `ExecutorService`? What implementations are available? Looking at the API documentation of `java.util.concurrent.Executors`[36] gives us a good idea of all the possibilities.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 7.7 (Hard)

Take a look through the various static methods in `Executors` to get a feel for the different implementations of `ExecutorService` that exist. Then, before continuing, go back and revisit your implementation of `fork` and try to find a counterexample, or convince yourself that the law holds for your implementation.

SIDEBAR Why laws about code and proofs are important

It may seem unusual to state and prove properties about an API. This certainly isn't something typically done in ordinary programming. Why is it important in Functional Programming?

In FP it's easy and expected to factor out common functionality into generic, reusable components that can be *composed*. Side effects hurt compositionality, but more generally, any hidden non-deterministic behavior that prevents us from treating our components as black boxes makes composition difficult or impossible.

A good example is our description of the law for `fork`. We can see that if the law we posited didn't hold, many of our general-purpose combinators that depend on `fork` such as `parMap` would no longer be sound. As a result, their usage might be dangerous since they could result in deadlocks when used in broader parallel computations.

Giving our APIs an algebra with meaningful laws that aid reasoning makes them more usable for clients. It also means that we can confidently treat all the objects of our APIs as black boxes. As we'll see in part 3, this is crucial for our ability to factor out common patterns across the different libraries we've written.

Putting on that testing hat, we write an assertion function to validate equality of two `Par` instances given `ExecutorService`. It is added as an infix `shouldBe` extension method on `Par`:

```
infix fun <A> Par<A>.shouldBe(other: Par<A>) = { es: ExecutorService ->
    if (this(es).get() != other(es).get())
        throw AssertionError("Par instances not equal")
}
```

Using this handy new assertion method, we'll discover a rather subtle problem that will occur in most implementations of `fork`. When using an `ExecutorService` backed by a thread pool of bounded size (see `Executors.newFixedThreadPool`), it's very easy to run into a deadlock. Suppose we have an `ExecutorService` backed by a thread pool where the maximum number of threads is 1. A deadlock will occur if we attempt to run the following example using our current implementation:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
val es = Executors.newFixedThreadPool(1)

val a: Par<Int> = lazyUnit { 42 + 1 }
val b: Par<Int> = fork { a }
(a shouldBe b)(es)
```

Can you see why this is the case? Let's take a closer look at our implementation of fork:

```
fun <A> fork(a: () -> Par<A>): Par<A> =
    { es ->
        es.submit(Callable<A> {
            a()(es).get()   ❶
        })
    }
```

❶ Waiting for the result of once `Callable` inside another `Callable`

We're submitting the `Callable` first, and *within that* `Callable`, we're submitting another `Callable` to the `ExecutorService` and blocking on its result. Recall that `a()(es)` will submit a `Callable` to the `ExecutorService` and get back a `Future`. This is a problem if our thread pool has size 1. The outer `Callable` gets submitted and picked up by the sole thread. Within that thread, before it will complete, we submit and block waiting for the result of another `Callable`. But there are no threads available to run this `Callable`. They're waiting on each other and therefore our code deadlocks.

EXERCISE 7.8 (Hard)

Show that any fixed-size thread pool can be made to deadlock given this implementation of `fork`.

When you find counterexamples like this, you have two choices—you can try to fix your implementation such that the law holds, or you can refine your law to state the conditions under which it holds more explicitly. For example, you could simply stipulate that you require thread pools that can grow unbounded. Even this is a good exercise. It forces you to document invariants or assumptions that were previously implicit.

Can we fix `fork` to work on fixed-size thread pools? Now let's look at a different implementation:

```
fun <A> fork(pa: () -> Par<A>): Par<A> =
    { es -> pa()(es) }
```

This certainly avoids deadlock. The only problem is that we aren't *actually* forking a separate logical thread to evaluate `pa` at all. So `fork(hugeComputation)(es)` for some `ExecutorService` would run `hugeComputation` in the main thread, which is exactly what we wanted to avoid by calling `fork`. Even though this is not the intention of `fork`, this is still a useful combinator since it lets us delay instantiation of a computation until it's actually needed. Let's give it a new name, `delay`:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> delay(pa: () -> Par<A>): Par<A> =
    { es -> pa()(es) }
```

What we'd really like to do is run arbitrary computations over fixed-size thread pools. In order to do that, we'll need to pick a different representation of `Par`.

7.4.3 Using actors for a non-blocking implementation

In this section, we'll develop a fully non-blocking implementation of `Par` that works for fixed-size thread pools. Since this isn't essential to our overall goals of discussing various aspects of functional design, you may skip to section 7.5 if you prefer. Otherwise, read on.

The essential problem with the current representation is that we can't get a value out of a `Future` without the current thread blocking on its `get` method. A representation of `Par` that doesn't leak resources this way has to be *non-blocking* in the sense that the implementations of `fork` and `map2` must never call a method that blocks the current thread like `Future.get` does. Writing such an implementation correctly can be challenging. Fortunately we have laws with which to test our implementation, and we only have to get it right *once*. After that, the users of our library can enjoy a composable and abstract API that does the right thing every time.

In the code that follows, you don't need to understand what's going on every step of the way. We just want to demonstrate, using real code, what a correct law abiding representation of `Par` might look like.

RETHINKING PAR AS NON-BLOCKING BY REGISTERING A CALLBACK

So how can we implement a non-blocking representation of `Par`? The idea is simple. Instead of turning `Par` into a `java.util.concurrent.Future` which only allows us to `get` a value through a blocking call, we'll introduce our own version of `Future`. Our version is able to *register a callback that will be invoked when the result is ready*. This is a slight shift in perspective.

```
abstract class Future<A> {
    internal abstract fun invoke(cb: (A) -> Unit)   ❶
}

typealias Par<A> = (ExecutorService) -> Future<A>   ❷
```

❶ The `invoke` method is declared `internal` so is not visible beyond our module

❷ `Par` looks the same as before, although we're using our new non-blocking `Future` instead of the one in `java.util.concurrent`.

Our brand new `Par` type looks identical to our initial representation, except that we're now returning `Future` which has a different API than that found in `java.util.concurrent.Future`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

. Rather than calling `get` to obtain the result from `Future`, our `Future` has an `invoke` method that receives a function `cb` which expects the result of type `A` and uses it to perform some effect. This kind of function is sometimes called a *continuation* or a *callback*.

The `invoke` method is marked `internal` so that we don't expose it to users of our library. Marking it `internal` restricts access of the method beyond the scope of our module. This is so that our API remains pure and we can guarantee that our laws are upheld.

SIDEBAR **Using local side effects for a pure API**

The `Future` type we defined here is rather imperative. The definition `(A) -> Unit` immediately raises eyebrows. Such a function can only be useful for executing some side effect using the given `A`, as we certainly won't be using the returned value `Unit`. Are we still doing functional programming in using a type like `Future`? Yes, although we're making use of a common technique of using side effects as an implementation detail for a purely functional API. We can get away with this because the side effects we use are *not observable* to code that uses `Par`. Note that the `invoke` method is `internal` and isn't even visible beyond our library.

As we go through the rest of our implementation of the non-blocking `Par`, you may want to convince yourself that the side effects employed can't be observed by external code. The notion of local effects, observability, and subtleties of our definitions of purity and referential transparency are discussed in much more detail in chapter 14, but for now an informal understanding is fine.

Let's begin by looking at an example of the actual creation of a `Par`. The simplest way to do this is through `unit`.

Listing 7.10 Creating a new non-blocking `Par` through `unit`

```
fun <A> unit(a: A): Par<A> =
    { es: ExecutorService ->
        object : Future<A>() {
            override fun invoke(cb: (A) -> Unit) = cb(a)   ❶
        }
    }
```

❶ Simply pass the value to the continuation. Done!

Since `unit` already has a value `a` of type `A` available, all this function needs to do is call the continuation `cb` passing in that value.

With this representation of `Par`, let's look at how we might implement the `run` function, which we'll change to just return an `A`. Since it goes from `Par<A>` to `A`, it will have to construct a continuation and pass it to the `Future` value's `invoke` method. By making this continuation

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

originate from here, it will release the latch and make the result available immediately.

Listing 7.11 Implementation of `run` method to accommodate non-blocking `Par`

```
fun <A> run(es: ExecutorService, pa: Par<A>): A {
    val ref = AtomicReference<A>()  ❶
    val latch = CountDownLatch(1)  ❷
    pa(es).invoke { a: A ->
        ref.set(a)
        latch.countDown()  ❸
    }
    latch.await()  ❹
    return ref.get()  ❺
}
```

❶ Create a mutable, thread-safe reference to store the result.

❷ A `CountDownLatch` blocks threads until countdown reaches 0

❸ Set the result and release the latch when result is received.

❹ Wait until the result is available and the latch is released.

❺ Once we've passed the latch, we know `ref` has been set and we return its value.

In our current implementation, `run` blocks the calling thread while waiting for the `latch` to be released. In fact, it isn't possible to write an implementation of `run` that *doesn't* block. Our method has to wait for a value of `A` to materialize before it can return anything. For this reason, we want users of our API to avoid calling `run` until they definitely want to wait for a result. We could even go so far as to remove `run` from our API altogether and expose the `invoke` method on `Par` so that users can register asynchronous callbacks. That would certainly be a valid design choice, but we'll leave our API as it is for now.

If the `latch` needs to be released only once, the function above can be simplified using a `CompletableFuture`. The `CompletableFuture` class is a non-abstract implementation of interface `Future` that is part of the JDK since Java 8. It gives the developer full control over making a result available while including all the thread blocking management provided by any `Future` implementation returned by `ExecutorService` methods.

Listing 7.12 Implementation of `run` method using a `CompletableFuture`

```
fun <A> run2(es: ExecutorService, pa: Par<A>): A {
    val ref = CompletableFuture<A>()  ❶
    pa(es).invoke { a: A ->
        ref.complete(a)  ❷
    }
    return ref.get()  ❸
}
```

❶ Create a `CompletableFuture` to manage blocking the current thread and store the result.

❷ Set the result. This unlocks the `CompletableFuture` and makes its result available.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❸ Wait until the result is available, then return its value.

As can be seen from this snippet, a `CountDownLatch` is not necessary anymore since blocking the thread is managed by the `CompletableFuture`.

We've already seen `unit` in listing 7.10, but What about `fork`? This is where we introduce the actual parallelism:

Listing 7.13 The `fork` function forks off a task to evaluate the lazy argument

```
fun <A> fork(a: () -> Par<A>): Par<A> =
    { es: ExecutorService ->
        object : Future<A>() {
            override fun invoke(cb: (A) -> Unit) =
                eval(es) { a()(es).invoke(cb) }   ❶
        }
    }

fun eval(es: ExecutorService, r: () -> Unit) {
    es.submit(Callable { r() })   ❷
}
```

❶ Call to `eval` forks off evaluation of `a` and returns immediately.

❷ A helper function to evaluate an action asynchronously using some `ExecutorService`.

When the `Future` returned by `fork` receives its continuation `cb`, it will fork off a task to evaluate the lazy argument `a`. Once the argument has been evaluated and called to produce a `Future<A>`, we register `cb` to be invoked when that `Future` has its resulting `A`.

Let's consider `map2`. Recall the signature for this combinator.

```
fun <A, B, C> map2(pa: Par<A>, pb: Par<B>, f: (A, B) -> C): Par<C>
```

Here, a non-blocking implementation is considerably trickier. Conceptually, we'd like `map2` to run both `Par` arguments in parallel. When both results have arrived, we want to invoke `f`, then pass the resulting `C` to the continuation. But there are several race conditions to worry about here, and a correct non-blocking implementation is difficult using only low-level primitives like those provided in `java.util.concurrent`.

A BRIEF DETOUR DEMONSTRATING THE USE OF ACTORS

To implement `map2`, we'll use a non-blocking concurrency primitive called *actors*. An `Actor` is essentially a concurrent process that doesn't constantly occupy a thread. Instead, it only occupies a thread when it receives a *message*. Importantly, although multiple threads may be concurrently sending messages to an actor, the actor processes only one message at a time, queueing other messages for subsequent processing. This makes them useful as a concurrency primitive when writing tricky code that must be accessed by multiple threads, and which would otherwise be prone to race conditions or deadlocks.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

It's best to illustrate this with an example. Many implementations of actors would suit our purposes just fine, but in the interest of simplicity we'll use our own minimal actor implementation included with the chapter code in the file `actor.kt`. We'll interact with it through the use of some client code to get a feel of how it works. We begin by getting the actor up and running.

Listing 7.14 Setting up an actor to handle client requests

```
val es: ExecutorService = Executors.newFixedThreadPool(4)   ❶
val s = Strategy.from(es)   ❷
val echoer = Actor<String>(s) {   ❸
    println("got message: $it")
}
```

❶ Create an `ExecutorService` instance `es` to back our actor

❷ Wrap `es` in a `Strategy` named `s`

❸ Spin up an actor using the `Strategy`, passing it an anonymous handler function.

Now that we have the instance of the actor referenced by `echoer`, we can send some messages.

```
echoer.send("hello")   ❶
//got message: hello   ❷

echoer.send("goodbye")   ❸
//got message: goodbye

echoer.send("You're just repeating everything I say, aren't you?")
//got message: You're just repeating everything I say, aren't you?
```

❶ Send the "hello" message to the actor.

❷ The spawned process goes off and invokes the handler, immediately freeing up the current thread to process next messages.

❸ Actor is sent new "goodbye" message without waiting for "hello" handler to complete.

It's not essential to understand the `Actor` implementation. A correct and efficient implementation is rather subtle, but if you're curious, see the `actor.kt` file in the chapter code. The implementation is under 100 lines of ordinary Kotlin code. The hardest part in understanding an actor implementation is the fact that multiple threads may be messaging the actor simultaneously. The implementation needs to ensure that messages are processed one at a time, and also that all messages sent to the actor will eventually be processed, rather than being queued indefinitely. Even so, the code ends up being concise.

IMPLEMENTING MAP2 VIA ACTORS

We can now implement `map2` using an `Actor` to collect the result from both arguments. The code is fairly straightforward, and there are no race conditions to worry about since we know that the `Actor` will only process one message at a time.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A, B, C> map2(pa: Par<A>, pb: Par<B>, f: (A, B) -> C): Par<C> =
    { es: ExecutorService ->
        object : Future<C>() {
            override fun invoke(cb: (C) -> Unit) {
                val ar = AtomicReference<Option<A>>(None)   ❶
                val br = AtomicReference<Option<B>>(None)
                val combiner =
                    Actor<Either<A, B>>(Strategy.from(es)) { eab ->   ❷
                        when (eab) {
                            is Left<A> ->   ❸
                                br.get().fold(
                                    { ar.set(Some(eab.a)) },
                                    { b -> eval(es) { cb(f(eab.a, b)) } }
                                )
                            is Right<B> ->   ❹
                                ar.get().fold(
                                    { br.set(Some(eab.b)) },
                                    { a -> eval(es) { cb(f(a, eab.b)) } }
                                )
                        }
                    }
                pa(es).invoke { a: A -> combiner.send(Left(a)) }   ❺
                pb(es).invoke { b: B -> combiner.send(Right(b)) }
            }
        }
    }
```

❶ Two `AtomicReference` instances are used to store mutable results

❷ An actor that awaits both results, combines them with `f`, and passes the result to `cb`

❸ Branch taken when a `Left(a)` is received, combines if a `Right(b)` was previously set in `br`.

❹ Branch taken when a `Right(b)` is received, combines if a `Left(a)` was previously set in `ar`

❺ Passes the actor as a continuation to both sides.

We have four possible scenarios to be dealt with in the `combiner` actor. Let's look at each one in turn:

- If the `A` result arrives first, it is stored in `ar` and the actor now waits for `B` to arrive.
- If the `A` result arrives last and `B` is already present, the results `a` and `b` are combined by `f` to be of type `C` and passed into the callback `cb`.
- If the `B` result arrives first, it is stored in `br` and the actor now waits for `A` to arrive.
- If the `B` result arrives last and `A` is already present, the results `a` and `b` are combined by `f` to be of type `C` and passed into the callback `cb`.

The actor is then passed as a continuation to both sides. It is wrapped as a `Left` in the case that it's an `A`, and `Right` when it's a `B`. We use the `Either` data type, invoking `Left(a)` and `Right(b)` constructors for each side of this union. They serve to indicate to the actor where the result originated.

Given these implementations, we should now be able to run `Par` values of arbitrary complexity without having to worry about running out of threads, even if the actors only have access to a single JVM thread.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

We can now write some client code to try out this fancy new machinery.

```
val p: (ExecutorService) -> Future<List<Double>> =
    parMap((1..10).toList()) { sqrt(it.toDouble()) }

val x: List<Double> =
    run(Executors.newFixedThreadPool(2), p)

println(x)
```

Running this code will yield a result as follows:

```
[1.0, 1.4142135623730951, 1.7320508075688772, 2.0, 2.23606797749979,
2.449489742783178, 2.6457513110645907, 2.8284271247461903, 3.0,
3.1622776601683795...
```

That will call `fork` about 100,000 times, starting that many actors to combine these values two at a time. Thanks to our non-blocking `Actor` implementation we don't need 100,000 JVM threads to perform this processing, but managed to do it with a fixed thread pool size of 2!

And so we have proved that our law of forking now holds for fixed-size thread pools.

EXERCISE 7.9 (Hard/Optional)

Our non-blocking representation doesn't currently handle errors at all. If at any point our computation throws an exception, the `run` implementation's `latch` never counts down and the exception is simply swallowed. Can you fix that?

Taking a step back, the purpose of this section hasn't necessarily been to figure out the best non-blocking implementation of `fork`, but more to show that laws are important. They give us another angle to consider when thinking about the design of a library. If we hadn't tried writing out some of the laws of our API, we may not have discovered the thread resource leak in our first implementation until much later.

In general, there are multiple approaches you can consider when choosing laws for your API. You can think about your conceptual model, and reason from there to postulate laws that should hold true. You can also just *invent* laws you think might be useful or instructive (like we did with our `fork` law), and see if it's possible and even sensible to ensure that they hold for your model. And lastly, you can look at your *implementation* and come up with laws you expect to hold based on that.[37]

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

7.5 Refining combinators to their most general form

Functional design is an iterative process. After you write your API and have at least a prototype implementation, try using it for progressively more complex or realistic scenarios. Sometimes you'll find that these scenarios require new combinators. But before jumping right to implementation of new combinators, it's a good idea to see if you can refine the combinator you need to *its most general form*. It may be that what you need is just a specific case of some more general combinator.

> **NOTE** For the sake of simplicity we will revert to using our original, simpler blocking representation of `Par<A>` instead of the newer non-blocking actor based solution. Feel free to attempt the exercises in this section using the non-blocking variant of `Par<A>`.

Let's look at an example of this generalization. Suppose we want a function to choose between two forking computations based on the result of an initial computation:

```
fun <A> choice(cond: Par<Boolean>, t: Par<A>, f: Par<A>): Par<A>
```

This constructs a computation that proceeds with `t` if `cond` results in `true`, or `f` if `cond` results in `false`. We can certainly implement this by blocking on the result of the `cond`, and then using this result to determine whether to run `t` or `f`. Here's a simple blocking implementation.

```
fun <A> choice(cond: Par<Boolean>, t: Par<A>, f: Par<A>): Par<A> =
    { es: ExecutorService ->
        when (run(es, cond).get()) {   ❶
            true -> run(es, t)
            false -> run(es, f)
        }
    }
```

❶ Block on the predicate `Par<Boolean>` before proceeding

But before we get satisfied and move on, let's think about this combinator a bit further. What is it doing? It's running `cond`, and then when the result is available, it runs either `t` or `f`. This seems reasonable, but let's think of some possible variations that capture the essence of this combinator. There is something rather arbitrary about the use of `Boolean` and the fact that we're only selecting among *two* possible parallel computations, `t` and `f` in this combinator. Why just two? If it's useful to choose between two parallel computations based on the results of a first, it should certainly be useful to choose between *N* computations:

```
fun <A> choiceN(n: Par<Int>, choices: List<Par<A>>): Par<A>
```

Let's say that `choiceN` runs `n`, and then uses that to select a parallel computation from `choices`. This is a bit more general than `choice`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 7.10

Implement `choiceN`, followed by `choice` in terms of `choiceN`.

Let's take a step back and observe what we've done in this iteration. We've generalized our original combinator `choice` to `choiceN`, now capable of expressing `choice` as well as other use cases not supported by `choice`. Let's keep going to see if we can refine `choice` to an even more general combinator.

The combinator `choiceN` remains somewhat arbitrary. The choice of `List` seems overly specific. Why does it matter what sort of container we have? For instance, what if instead of a `List` we have a `Map` of computations? The `Map<K, V>` is a data structure that associates keys of type `K` with values of type `V`. The one-to-one relationship of `K` to `V` allows us to look up a value by its associated key.

EXERCISE 7.11

Implement a combinator called `choiceMap` that accepts a `Map<K, Par<V>>` as container.

```
fun <K, V> choiceMap(
    key: Par<K>,
    choices: Map<K, Par<V>>
): Par<V> = TODO()
```

TIP Don't be overly concerned with handling of null values returned by `Map.get`. For the sake of this exercise, consider using `Map.getValue` for value retrieval.

Even the `Map` encoding of the set of possible choices feels overly specific, just like `List` was. If we look at our implementation of `choiceMap`, we can see we aren't really using much of the API of `Map`. Really, the `Map<A,Par<B>>` is used to provide a function, `(A) -> Par<B>`. And now that we've spotted that, looking back at `choice` and `choiceN`, we can see that for `choice`, the pair of arguments was just being used as a function of type `(Boolean) -> Par<A>` (where the `Boolean` selects one of the two `Par<A>` arguments), and for `choiceN` the list was just being used as a function of type `(Int) -> Par<A>`!

Let's make a more general signature that unifies them all. We'll call it `chooser`, and allow it to perform selection through a function `(A) -> Par<B>`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 7.12

Implement this new primitive `chooser`, and then use it to implement `choice`, `choiceN` and `choiceMap`.

```
fun <A, B> chooser(pa: Par<A>, choices: (A) -> Par<B>): Par<B> = TODO()
```

Whenever you generalize functions like this, take a critical look at your final product. Although the function may have been motivated by some specific use case, the signature and implementation may have a more general meaning. In this case, `chooser` is perhaps no longer the most appropriate name for this operation, which is actually quite general—it's a parallel computation that, when invoked will run an initial computation whose result is used to determine a second computation. Nothing says that this second computation even needs to *exist* before the first computation's result is available. It doesn't event need to be stored in a container like `List` or `Map`. Perhaps it's being *generated* from whole cloth using the result of the first computation. This function, which comes up often in functional libraries, is usually called `bind` or `flatMap`:

```
fun <A, B> flatMap(pa: Par<A>, f: (A) -> Par<B>): Par<B>
```

Is `flatMap` really the most primitive possible function, or can we generalize it yet further? Let's play around a bit more. The name `flatMap` is suggestive of the fact that this operation could be decomposed into two steps: *mapping* `f: (A) -> Par<B>` over our `Par[A]`, which generates a `Par<Par<B>>`, and *flattening* this nested `Par<Par<B>>` to a `Par<B>`.

Here is the interesting part—it suggests that all we needed to do was add an *even simpler* combinator, let's call it `join`, for converting a `Par<Par<X>>` to `Par<X>` for *any* choice of `X`.

Again we're simply following the types. We have an example that demands a function with a given signature, and so we just bring it into existence. Now that it exists, we can think about what the signature means. We call it `join` since conceptually it's a parallel computation that, when run, will execute the inner computation, wait for it to finish (much like `Thread.join`), and then return its result.

EXERCISE 7.13

Implement `join`. Can you see how to implement `flatMap` using `join`? And can you implement `join` using `flatMap`?

```
fun <A> join(a: Par<Par<A>>): Par<A> = TODO()
```

We'll stop here, although you're encouraged to explore this algebra further. Try more complicated examples, discover new combinators, and see what you find! If you are so inclined,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

here are some questions to consider:

- Can you implement a function with the same signature as `map2`, but using `flatMap` and `unit`? How is its meaning different than that of `map2`?
- Can you think of laws relating `join` to the other primitives of the algebra?
- Are there parallel computations that can't be expressed using this algebra? Can you think of any computations that can't even be expressed by adding new primitives to the algebra?

SIDEBAR **Recognizing the expressiveness and limitations of an algebra**

As you practice more functional programming, one of the skills you'll develop is the ability to recognize *what functions are expressible from an algebra*, and what the limitations of that algebra are. For instance, in the preceding example it may not have been obvious at first that a function like `choice` couldn't be expressed purely in terms of `map`, `map2`, and `unit`. It may also not have been obvious that `choice` was just a special case of `flatMap`. Over time, observations like this will come quicker, and you'll also get better at spotting how to modify your algebra to make some needed combinator expressible. These skills will be helpful for all of your API design work.

Being able to reduce an API to a minimal set of primitive functions is an extremely useful skill. It often happens that primitive combinators encapsulate some tricky logic, and reusing them means we don't have to duplicate our work.

We've now completed the design of a library for defining parallel and asynchronous computations in a purely functional way. Although this domain is interesting, the primary goal of this chapter was to give you a window into the process of functional design, a sense of the kind of issues you're likely to encounter, and ideas on how to handle such issues.

Chapters 4 through 6 continually referred to the principle of *separation of concerns*. Specifically, the idea of separating the *description* of a computation from the *interpreter* that runs it. In this chapter we saw this principle in action—we designed a library that describes parallel computations as values of a data type `Par`, with a separate interpreter called `run` to spawn threads to execute them.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

7.6 Summary

- Functional API design is an iterative and exploratory process driven by real-world examples.
- A purely functional library that deals with parallelization is a perfect example to demonstrate API design.
- *Data types* and their associated *functions* are born out of exploring domain examples.
- Treating an API as you would an *algebraic equation* leads to following types to a concrete implementation.
- Laws help define constraints on operations, lead to implementation choices and validate properties.
- Combinators can often be generalized to broaden their application across many different applications and scenarios.
- Effective library design separates description of computations from the interpreter that will be responsible for running them.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

8 Property-based testing

This chapter covers:

- Understanding the concept of property-based testing
- Model properties by discovering appropriate data types and functions
- Fabricating test data using generators
- Minimizing test case outcomes to give meaningful feedback
- Using properties to affirm laws
- Applying syntactic sugar to improve user experience

In chapter 7 we worked through the design of a functional library for expressing parallel computations. We introduced the idea that an API should form an *algebra*—that is, a collection of data types, functions over these data types, and importantly, *laws* or *properties* that express relationships between these functions. We also hinted at the idea that it might be possible to somehow *validate* these laws automatically. Validation is an important step, as we need to know that the code we write is in conformance with the laws that we have imposed upon our program. It would be of great benefit if we could somehow automate this validation process.

This chapter will take us toward a simple but powerful library for automated *property-based testing*. The general idea of such a library is to decouple the specification of program behavior from the creation of test cases. The programmer focuses on specifying the behavior of a program and giving high-level constraints on the test cases. The framework then automatically generates test cases that satisfy these constraints and runs tests to validate that the program behaves as specified.

Although a library for testing has a very different purpose than a library for parallel computations, surprisingly we'll discover that they both have very similar combinators. This similarity is something we'll return to again in part 3.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

8.1 A brief tour of property-based testing

Property-based testing frameworks are already broadly accepted and used among functional programmers in many different languages such as Haskell, Scala, and even in Kotlin. As an example, let's look at KotlinTest, a popular testing framework for Kotlin development. It has built-in support for property-based testing in which a property looks something like this.

Listing 8.1 Demonstration of property-based testing using KotlinTest

```
val intList = Gen.list(Gen.choose(0, 100))   ❶

forAll(intList) {   ❷
    (it.reversed().reversed() == it) and   ❸
            (it.firstOption() == it.reversed().lastOrNone())   ❹
}

forAll(intList) {   ❺
    it.reversed() == it
}
```

❶ A generator of lists containing integers between 0 and 100.

❷ A valid property that specifies the behavior of the `List.reversed` method.

❸ Check that reversing a list twice gives back the original list.

❹ Check that the first element becomes the last element after reversal.

❺ Second property that fails under most conditions

Here, `intList` is not a `List<Int>` as you might expect, but rather a `Gen<List<Int>>`, which is something that knows how to generate test data of type `List<Int>`. We can *sample* from this generator to produce lists of different lengths, each filled with random numbers between 0 and 100. Generators in a property-based testing library have a rich API. We can combine and compose generators in different ways, reuse them, and so on.

The function `forAll` creates a *property* by combining a generator of type `Gen<A>` with some predicate of type `(A) -> Boolean`. The property asserts that all values produced by the generator should satisfy this predicate. Like generators, properties should also have a rich API.

Although KotlinTest does not currently support this, we should be able to use operators like `and` and `or` to combine multiple properties. The resulting property would hold only if none of the properties could be *falsified* by any of the generated test cases. Together, these combined properties would form a complete specification of the correct behavior to be validated.

It is well worth noting that the goal of this sort of testing is not necessarily to fully specify program behavior, but rather to give greater confidence in the code. Property-based testing does *not* replace unit testing, which finds its purpose more in expressing intent and driving design than in validating our confidence in the code.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

When we express these properties, KotlinTest will randomly generate `List<Int>` values to try to find a case that falsifies the predicates that we've supplied. It generates 100 test cases (of type `List<Int>`) and each list will be checked to see if it satisfies the predicates. Properties can of course fail—the second property should indicate that the predicate tested false for some input, which is then printed to standard out to facilitate further testing or debugging.

EXERCISE 8.1

To get used to thinking about testing in this way, come up with properties that specify the implementation of a `sum: (List<Int>) -> Int` function. You don't have to write your properties down as executable KotlinTest code—an informal description is fine. Here are some ideas to get you started:

- Reversing and summing a list should give the same result as summing the original, non-reversed list.
- What should the sum be if all elements of the list are the same value?
- Do any other properties spring to mind?

EXERCISE 8.2

What properties specify a function that finds the maximum of a `List<Int>`?

Property-based testing libraries often come equipped with other useful features. We'll talk more about some of these features later, but just to give an idea of what's possible:

- *Test case minimization* — In the event of a failing test, the framework tries increasingly smaller dataset sizes until it finds the *smallest* dataset that still fails, this being more illuminating for diagnosing failures. For instance, if a property fails for a list of size 10, the framework tries smaller lists and reports the smallest list that fails the test.
- *Exhaustive test case generation* — We call the set of values that could be produced by some `Gen<A>` the *domain*.[38] When the domain is small enough (for instance, if it's all even integers less than 100), we may exhaustively test all its values rather than generate sample values. If the property holds for all values in a domain, we have an actual *proof* rather than just the absence of evidence to the contrary.

KotlinTest is just one framework that provides property-based testing capabilities. And while there's nothing wrong with it, we'll derive our own library in this chapter, starting from scratch. Like in chapter 7, this is mostly for pedagogical purposes, but also partly because we should consider no library to be the final word on any subject. There is certainly nothing wrong with using an existing library like KotlinTest, and existing libraries can be a good source of ideas. But even if you decide you like the existing library's solution, spending an hour or two playing with designs and writing down some type signatures is a great way to learn more about the domain and understand the design trade-offs.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

8.2 Choosing data types and functions

This section will be another somewhat messy and iterative process of discovering data types and functions for our library. This time around we're designing a library for property-based testing in order to validate the laws or properties of our programs. As before, this is a chance to peek over the shoulder of someone working through possible scenarios and designs.

The particular path we take and the library we arrive at isn't necessarily the same as what you would come up with on your own. If property-based testing is unfamiliar to you, even better; this is a chance to explore a new domain and its design space while making your own discoveries about it. If at any point you are feeling inspired or have ideas of your own about how to design a library like this, don't wait for an exercise to prompt you—*put the book down* and explore your ideas. You can always come back to this chapter if you run out of ideas or get stuck.

8.2.1 Gathering initial snippets for a possible API

With that said, let's get started. Whenever we begin with library design, we need to define some data types that embody the concepts of our library. With this starting point in mind, what data types should we use for our testing library? What primitives should we define, and what might they mean? What laws should our functions satisfy? As before, we can look at a simple example and "read off" the needed data types and functions and see what we find. For inspiration, let's look at the KotlinTest example we showed earlier:

```
val intList = Gen.list(Gen.choose(0, 100))

forAll(intList) {
    (it.reversed().reversed() == it) and
            (it.firstOption() == it.reversed().lastOrNone())
}
```

Without knowing anything about the implementation of `Gen.choose` or `Gen.list`, we can guess that whatever data type they return (let's call it `Gen`, short for *generator*) must be parametric in some type. That is, `Gen.choose(0,100)` probably returns a `Gen<Int>`, and `Gen.list` is then a function with the signature `(Gen<Int>) -> Gen<List<Int>>`. But since it doesn't seem like `Gen.list` should care about the type of the `Gen` it receives as input. It would be odd to require separate combinators for creating lists of `Int`, `Double`, `String` and so on, so let's go ahead and make it polymorphic. We'll call our method `listOf` to make the intent clear.

```
fun <A> listOf(a: Gen<A>): List<Gen<A>> = TODO()
```

We can learn many things by looking at this signature. Notice that we're not specifying the size of the list to generate. For this to be implementable, our generator must therefore either make an assumption, or otherwise be told the size explicitly. Assuming a size seems a bit inflexible, any assumption is unlikely to be appropriate in all contexts. So it seems that generators must be told the size of test cases to generate. We can imagine an API where this is made explicit:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> listOfN(n: Int, a: Gen<A>): List<Gen<A>> = TODO()
```

This would certainly be a useful combinator, but not having to explicitly specify sizes is powerful as well. It means that whatever function runs the tests has the freedom to choose test case sizes, which opens up the possibility of doing the test case minimization we mentioned earlier. If the sizes are always fixed and specified by the programmer, the test runner won't have this flexibility. Keep this concern in mind as we get further along in our design.

What about the rest of this example? The `forAll` function looks interesting. We can see that it accepts a `Gen<List<Int>>` and what looks to be a corresponding predicate, `(List<Int>) -> Boolean`. But again, it doesn't seem like `forAll` should care about the types of the generator and the predicate, as long as they match up. We can express this with the type:

```
fun <A> forAll(a: Gen<A>, f: (A) -> Boolean): Prop = TODO()
```

Here, we've simply invented a new type, `Prop` (short for *property*) for the result of binding a `Gen` to a predicate. We might not know the internal representation of `Prop` or what other functions it supports, but based on our prior discussion in section 8.1, it should be combined with other `Prop` instances through the use of an `and` method. Let's introduce that as a new interface:

```
interface Prop {
    fun and(p: Prop): Prop
}
```

8.2.2 Exploring the meaning and API of properties

Now that we have a few fragments of an API, let's discuss what we want our types and functions to entail. First, let's consider `Prop`. We know of functions `forAll` (for creating a property), `and` (for composing properties), and will now learn about `check`. Here we deviate further from KotlinTest's property design as it doesn't have such a method in its API. We'll imagine this to be a method that *runs* our property and has a side effect of printing to the console. We will expose this as a convenience function on `Prop`, giving it a return type of `Unit` for now.

```
interface Prop {
    fun check(): Unit
    fun and(p: Prop): Prop
}
```

This return type does raise an issue in that we can't chain together multiple `checked` `Props` using the `and` operator. This might remind you of a similar problem that we encountered in chapter 7 when we looked at using `Thread` and `Runnable` for parallelism.

Since `check` has a side effect, the only option for implementing `and` in this case would be to run `check` on both `Prop` instances. So if `check` prints out a test report we would get two of them,

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

each printing failures and successes independently of each other. That's likely not the correct outcome. The problem is not so much that `check` has a side effect, but more generally that it throws away information by returning `Unit`.

In order to combine `Prop` values using combinators like `and`, we need `check` (or whatever function "runs" properties) to return some meaningful value. What type should that value have? Well, let's consider what sort of information we'd expect to get out of checked properties. As a minimum, we need to know whether the property succeeded or failed, so a `Boolean` return value would do just fine as a first pass. We now have enough to go ahead with implementing the `and` method.

EXERCISE 8.3

Assuming the following representation, use `check` to implement `and` as a method of `Prop`.

```
interface Prop {
    fun check(): Boolean
    fun and(p: Prop): Prop = TODO()
}
```

In this representation, `Prop` is nothing more than a non-strict `Boolean`. Any of the usual `Boolean` functions (AND, OR, NOT, XOR, and so on) can easily be defined for `Prop`. But a `Boolean` alone is probably insufficient. If a property fails, we might want to know how many tests succeeded first. We might also be interested in what arguments produced the failure. And if a property succeeds, it would be useful to know how many tests it ran. Next, let's encode this information by returning an `Either` to indicate success or failure:

```
typealias SuccessCount = Int

interface Prop {
    fun check(): Either<String, SuccessCount>
    fun and(p: Prop): Prop
}
```

For now, we've assigned the failure case to be `String`, but what type *should* we return on the left side? We don't know anything about the type of the test cases being generated. Should we add a type parameter to `Prop` and make it `Prop<A>` so `check` could return `Either<A, SuccessCount>`. Before going too far down this route, let's ask ourselves whether we really care about the *type* of the value that caused the property to fail. We don't really. We would only care about the type if we were going to do further computation with the failure.

NOTE We prefer using type aliases instead of simple types like `String`, `Int` or `Double` because we can assign meaningful names to them. This makes our code far easier to comprehend by others who will interact with it.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Most likely we're just going to end up printing it to the screen for inspection by whoever runs the tests. After all, the goal here is to find bugs and to indicate test cases that triggered them so they can be fixed. As a general rule, we shouldn't use `String` to represent data that we want to compute with. But for values that we're just going to show to human beings, a `String` is absolutely appropriate. This suggests that we can get away with the following representation for `Prop`:

```
typealias SuccessCount = Int
typealias FailedCase = String

interface Prop {
    fun check(): Either<Pair<FailedCase, SuccessCount>, SuccessCount>
    fun and(p: Prop): Prop
}
```

In the case of failure, `check` returns a `Left(Pair(s,n))`, where `s` is some `String` that represents the value that caused the property to fail, and `n` is the number of cases that succeeded before the failure occurred. Conversely, a success would be a `Right(n)` where `n` represents the total number of cases succeeded.

For now, that takes care of the return value of `check`, but what about its arguments? Right now, the `check` method takes none. Is this sufficient? Since `check` is a method on `Prop`, we can think about what information is available to it at the time of its creation. In particular, let's take another look at `forAll`:

```
fun <A> forAll(a: Gen<A>, f: (A) -> Boolean): Prop = TODO()
```

Without knowing more about the representation of `Gen`, it is hard to say whether there is enough information here to be able to generate values of type `A`. Why is this important? We will need this information to implement `check`. So for now, we'll take a step back and turn our attention to `Gen` to get a better idea of what it means, and what its dependencies might be.

8.2.3 Discovering the meaning and API of generators

We determined earlier that a `Gen<A>` was something that knows how to generate values of type `A` . How could it go about doing this? Well, it could *randomly* generate these values. Considering that we devoted all of chapter 6 to this topic, it would seem like we're missing a trick if we don't use what we learned in that chapter! Thinking back to our example, we provided an interface for a purely functional random number generator, `RNG`. We then showed how to make it convenient to combine computations that make use of it. If we recall the definition of `State`, we can simply make `Gen` a type that wraps a `State` transition over a random number generator.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 8.2 Define `Gen` by wrapping a state transition over a random number generator

```
interface RNG {
    fun nextInt(): Pair<Int, RNG>
}

data class State<S, out A>(val run: (S) -> Pair<A, S>)

data class Gen<A>(val sample: State<RNG, A>)
```

EXERCISE 8.4

Implement `Gen.choose` using this representation of `Gen`. It should generate integers in the range `start` to `stopExclusive`. Feel free to use functions you've already written.

```
fun choose(start: Int, stopExclusive: Int): Gen<Int> = TODO()
```

EXERCISE 8.5

Let's see what else we can implement using this representation of `Gen`. Try implementing `unit`, `boolean`, and `listOfN` with the following signatures, once again drawing on functions previously written:

```
fun <A> unit(a: A): Gen<A> = TODO()

fun boolean(): Gen<Boolean> = TODO()

fun <A> listOfN(n: Int, ga: Gen<A>): Gen<List<A>> = TODO()
```

As discussed in chapter 7, we're interested in understanding which operations are *primitive* and which are *derived*, and in finding a small yet expressive set of primitives. A good way to explore what is possible with a given set of primitives is to pick some concrete examples you'd like to express and see if you can assemble the functionality you want. As you do so, look for patterns, try factoring out these patterns into combinators, and refine your set of primitives. We encourage you to stop reading here and simply *play* with the primitives and combinators we've written so far. If you want some concrete examples to inspire you, here are a few ideas:

- If we can generate a single `Int` in some range, do we also need a new primitive to generate a `Pair<Int, Int>` in some range?
- Can we produce a `Gen<Option<A>>` from a `Gen<A>`? What about a `Gen<A>` from `Gen<Option<A>>`?
- Can we generate strings using our existing primitives?

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **The importance of play**

You don't have to wait around for a concrete example to explore the problem domain of your library. In fact, if you rely exclusively on such useful or important examples to design your API, you'll often miss out on crucial design aspects and end up writing APIs with overly specific features.

We don't want to *overfit* our design to the particular examples we happen to think of right now. We want to reduce the problem to its essence, and sometimes the best way to do this is by *playing*. Don't try to solve important problems or produce useful functionality. At least, not right away. Just experiment with different representations, primitives and operations. Let questions naturally arise and explore whatever piques your interest.

Observations like "These two functions seem similar. I wonder if there's some more general operation hiding inside," or "Would it make sense to make this data type polymorphic?", or "What would it mean to change this aspect of the representation from a single value to a `List` of values?" will begin to surface.

There is no right or wrong way to do this, but there are so many different design choices that it's impossible *not* to run headlong into fascinating questions to play with. It doesn't matter where you begin—if you keep playing, the domain will inexorably guide you to make all the design choices that are required.

8.2.4 Generators that depend on generated values

Suppose we'd like a `Gen<Pair<String, String>>` that generates pairs where the second string contains only characters from the first. Or that we had a `Gen<Int>` that chooses an integer between 0 and 11, and we'd like to make a `Gen<List<Double>>` that then generates lists of whatever length is chosen. In both of these cases there's a dependency—we generate a value, and then use that value to determine what generator to use next. For this we need `flatMap`, which lets one generator depend on another.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 8.6

Implement `flatMap`, and then use it to implement this more dynamic version of `listOfN`. Place `flatMap` and `listOfN` in the `Gen` data class as shown.

```
data class Gen<A>(val sample: State<RNG, A>) {

    companion object {
        fun <A> listOfN(gn: Gen<Int>, ga: Gen<A>): Gen<List<A>> = TODO()
    }

    fun <B> flatMap(f: (A) -> Gen<B>): Gen<B> = TODO()
}
```

EXERCISE 8.7

Implement `union` for combining two generators of the same type into one by pulling values from each generator with equal likelihood.

```
fun <A> union(ga: Gen<A>, gb: Gen<A>): Gen<A> = TODO()
```

EXERCISE 8.8

Implement `weighted`, a version of `union` that accepts a weight for each `Gen` and generates values from each `Gen` with probability proportional to its weight.

```
fun <A> weighted(
    pga: Pair<Gen<A>, Double>,
    pgb: Pair<Gen<A>, Double>
): Gen<A> = TODO()
```

8.2.5 Refining the property data type

Now that we have explored the representation of our generators, let's return to our definition of `Prop`. Our `Gen` representation has revealed information about the requirements for `Prop`. Our current definition of `Prop` looks like this, ignoring the `and` operator for now:

```
interface Prop {
    fun check(): Either<Pair<FailedCase, SuccessCount>, SuccessCount>
}
```

At this point `Prop` is nothing more than an `Either`, although it's missing some vital information. We have the number of successful test cases in `SuccessCount`, but we haven't specified how many test cases to examine before we consider the property to have *passed* the test. We could

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

certainly hardcode some value, but it would be far better to abstract over this detail. We will do so by injecting an integer aliased as `TestCases`. We will also turn `Prop` into a data class and make `check` a value instead of a method.

```
typealias TestCases = Int

typealias Result = Either<Pair<FailedCase, SuccessCount>, SuccessCount>

data class Prop(val check: (TestCases) -> Result)
```

Also, we're recording the number of successful tests on both sides of `Either`. But when a property passes, it's implied that the number of passed tests will be equal to the argument to `check`. So the caller of `check` learns nothing new by being told the success count. Since we don't currently need any information in the `Right` case of that `Either`, we can turn it into an `Option`:

```
typealias Result = Option<Pair<FailedCase, SuccessCount>>

data class Prop(val check: (TestCases) -> Result)
```

This now seems a bit weird since `None` will mean that all tests succeeded, and `Some` will indicate a failure. Until now, we've only ever used the `None` case of `Option` to indicate failure, but in this case we're using it to represent the *absence* of a failure. That is a perfectly legitimate use for `Option`, but its intent isn't very clear. So let's make a new data type equivalent to `Option<Pair<FailedCase, SuccessCount>>` that makes our intent more explicit.

Listing 8.3 Model the possible results of a test run as an ADT

```
sealed class Result {   ❶
    abstract fun isFalsified(): Boolean
}

object Passed : Result() {   ❷
    override fun isFalsified(): Boolean = false
}

data class Falsified(   ❸
    val failure: FailedCase,
    val successes: SuccessCount
) : Result() {
    override fun isFalsified(): Boolean = true
}
```

❶ Sealed type of `Result`

❷ Subtype indicates that all tests passed

❸ Subtype indicates that one of the test cases falsified the property

Is this a sufficient representation of `Prop` now? Let's take another look at `forAll`. Can `forAll` be implemented? If not, why not?

```
fun <A> forAll(a: Gen<A>, f: (A) -> Boolean): Prop = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

As we can see, `forAll` doesn't have enough information to return a `Prop`. Besides the number of test cases to try, `check` must have all the information needed to generate test cases. If it needs to generate random test cases using our current representation of `Gen`, it's going to need an `RNG`. Let's go ahead and supply this dependency to `Prop`:

Listing 8.4 Supply an instance of `RNG` for `Prop` to allow test case generation

```
data class Prop(val check: (TestCases, RNG) -> Result)
```

If we think of other dependencies that it might need besides the number of test cases and the source of randomness, we can just add these as extra parameters to `check` later.

By supplying `RNG` as parameter to `Prop`, we now have enough information available to implement `forAll`. Here's a first stab.

Listing 8.5 Implementation of `forAll` using all the underlying building blocks

```
fun <A> forAll(ga: Gen<A>, f: (A) -> Boolean): Prop =
    Prop { n: TestCases, rng: RNG ->
        randomSequence(ga, rng).mapIndexed { i, a ->   ❶
                try {
                    if (f(a)) Passed
                    else Falsified(a.toString(), i)   ❷
                } catch (e: Exception) {
                    Falsified(buildMessage(a, e), i)   ❸
                }
            }.take(n)
            .find { it.isFalsified() }
            .toOption()
            .getOrElse { Passed }
    }

private fun <A> randomSequence(
    ga: Gen<A>,
    rng: RNG
): Sequence<A> =   ❹
    sequence {
        val (a: A, rng2: RNG) = ga.sample.run(rng)
        yield(a)
        yieldAll(randomSequence(ga, rng2))
    }

private fun <A> buildMessage(a: A, e: Exception) =   ❺
    """
    |test case: $a
    |generated and exception: ${e.message}
    |stacktrace:
    |${e.stackTrace.joinToString("\n")}
"""".trimMargin()
```

❶ Prepare a `Sequence` of indexes `i` mapped to generated values `a`

❷ On test failure, record failed case and index exposing how many tests succeeded before failure.

❸ In the case of an exception, record as a result with pretty message.

❹ Generates infinite sequence of `A` recursively, sampling a generator.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❺ Use string interpolation and margin trim to build a pretty message.

Notice that we're catching exceptions and reporting them as test failures rather than letting `check` throw the exception. This is so that we don't lose information about what argument potentially triggered the failure.

NOTE We are using the Kotlin standard library `Sequence` type that allows us to generate a lazy stream of values by using the `sequence`, `yield` and `yieldAll` functions. The detail of this is not very important, and all we need to know is that we take `n` elements from the `Sequence` and apply a terminal operation to `find` an occurrence that is falsified, else we report a pass.

EXERCISE 8.9

Now that we have a representation of `Prop`, implement `and` and `or` for composing `Prop` values. Notice that in the case of an `or` failure, we don't know which property was responsible, the left or the right. Can you devise a way of handling this?

```
data class Prop(val run: (TestCases, RNG) -> Result) {
    fun and(p: Prop): Prop = TODO()

    fun or(p: Prop): Prop = TODO()
}
```

8.3 Test case minimization

Earlier, we mentioned the idea of test case minimization. By this we mean that we'd like our framework to find the smallest or simplest failing test case to better illustrate a failure and facilitate debugging. Let's see if we can tweak our representations to support this outcome. There are two general approaches we could take:

- *Shrinking* — After we've found a failing test case, we can run a separate procedure to minimize the test case by successively decreasing its "size" until it no longer fails. This is called *shrinking*, and it usually requires us to write separate code for each data type to implement this minimization process.
- *Sized generation* — Rather than shrinking test cases, we simply generate our test cases in order of *increasing* size and complexity. So we start small and increase the size until we find a failure. This idea can be extended in various ways to allow the test runner to make larger jumps in the space of possible sizes while still making it possible to find the smallest failing test.

KotlinTest, in addition to most of the popular property-based testing frameworks like ScalaCheck[39] and Haskell's QuickCheck[40] take the first approach of shrinking. Due to the greater complexity involved in implementing the approach, we'll choose to use the alternative

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

option instead. Sized generation is simpler, and in some ways more modular because our generators only need to know how to generate a test case of a given size. We'll see how this plays out shortly.

Instead of modifying our `Gen` data type for which we've already written a number of useful combinators, let's introduce sized generation as a separate layer in our library. A simple representation of a sized generator is just a function that takes a size and produces a generator:

Listing 8.6 Representation of a sized generator as a function from `Int` to `Gen`

```
data class SGen<A>(val forSize: (Int) -> Gen<A>)
```

EXERCISE 8.10

Implement a helper function called `unsized` for converting `Gen` to `SGen`. You can add this as a method on `Gen`.

```
data class Gen<A>(val sample: State<RNG, A>) {
    fun unsized(): SGen<A> = TODO()
}
```

EXERCISE 8.11

Not surprisingly, `SGen` at a minimum supports many of the same operations as `Gen`, and the implementations are rather mechanical. Define some convenience functions on `SGen` that simply delegate to the corresponding functions on `Gen`. Also provide a convenient way of invoking an `SGen`.

```
data class SGen<A>(val forSize: (Int) -> Gen<A>) {

    operator fun invoke(i: Int): Gen<A> = TODO()

    fun <B> map(f: (A) -> B): SGen<B> = TODO()

    fun <B> flatMap(f: (A) -> Gen<B>): SGen<B> = TODO()
}
```

NOTE Even though this approach is very repetitive, we will continue doing it this way for now. Part 3 of this book will present a better approach of handling such repetition.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 8.12

Implement a `listOf` combinator on `Gen` that doesn't accept an explicit size and should return an `SGen` instead of a `Gen`. The implementation should generate lists of the size provided to the `SGen`.

```
fun listOf(): SGen<List<A>> = TODO()
```

Next, let's see how `SGen` affects the definition of `Prop`, and in particular its `forAll` method. The `SGen` version of `forAll` looks like this:

```
fun <A> forAll(g: SGen<A>, f: (A) -> Boolean): Prop = TODO()
```

On closer inspection of this declaration, we see that it isn't possible to implement it. This is because `SGen` is expecting to be told a size, but `Prop` doesn't receive any such information. Much like we did with the source of randomness and number of test cases in the underlying `check` function of Prop (see Listing 8.4), we simply need to add this new number as a dependency to the function. So since we want to put `Prop` in charge of invoking the underlying generators with various sizes, we'll have `Prop` accept a *maximum* size. `Prop` will then generate test cases up to and including the maximum specified size. An addition benefit is that this will also allow it to search for the smallest failing test case. Let's see how this works out.

NOTE This rather simplistic implementation gives an equal number of test cases to each size being generated, and increases the size by `1` starting from `0`. We could imagine a more sophisticated implementation that does something like a binary search for a failing test case size—starting with sizes `0,1,2,4,8,16`…, and then narrowing the search space in the event of a failure.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 8.7 Generating test cases up to a given maximum size

```
typealias MaxSize = Int

data class Prop(val check: (MaxSize, TestCases, RNG) -> Result) {

    companion object {

        fun <A> forAll(g: SGen<A>, f: (A) -> Boolean): Prop =
            forAll({ i -> g(i) }, f)   ❶

        fun <A> forAll(g: (Int) -> Gen<A>, f: (A) -> Boolean): Prop =
            Prop { max, n, rng ->

                val casePerSize: Int = (n + (max - 1)) / max   ❶

                val props: Sequence<Prop> =
                    generateSequence(0) { it + 1 }   ❸
                        .take(min(n, max) + 1)
                        .map { i -> forAll(g(i), f) }   ❹

                val prop: Prop = props.map { p ->
                    Prop { max, _, rng ->
                        p.check(max, casePerSize, rng)
                    }
                }.reduce { p1, p2 -> p1.and(p2) }   ❺

                prop.check(max, n, rng)   ❻
            }

    }

    fun and(p: Prop): Prop =
        Prop { max, n, rng ->   ❼
            when (val prop = check(max, n, rng)) {
                is Passed -> p.check(max, n, rng)
                is Falsified -> prop
            }
        }
}
```

❶ The entry point that is used in tests

❷ Generate this many random cases for each size

❸ Generate an incrementing `Sequence<Int>` starting at `0`

❹ Make one property per size, but never more than `n` properties (uses previously defined `forAll`)

❺ Combine them all into one property using `Prop.and`.

❻ Check the combined property

❼ Retrofit `and` to handle new `max` parameter

This code might seem a bit daunting at first, but on closer examination it's quite straight forward. `check` now has a new `MaxSize` parameter which sets an upper bound to the size of test cases to run. Our `forAll` entry point takes an `SGen` and a predicate which is passed through to our new `forAll` function, and in turn generates a combined `Prop`.

This property first calculates the number of test cases to run per size. It then generates a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`Sequence` consisting of one `Prop` per size using the previously defined `forAll` function from Listing 8.5. Lastly, it combines them all into a single property using an updated version of our previously defined `and` function. At the end of all this, the remaining reduced property is checked.

8.4 Using the library and improving user experience

We've now converged on what seems like a reasonable API. We could keep tinkering with it, but at this point let's try *using* it instead. We will do this by constructing tests and seeing if we notice any deficiencies, either in what it can express or in its general usability. Usability is somewhat subjective, but we generally like to have convenient syntax and appropriate helper functions for common usage patterns. We aren't necessarily aiming to make the library more expressive, but we do want to make it pleasant to use.

8.4.1 Some simple examples

Let's revisit an example that we mentioned at the start of this chapter—specifying the behavior of a function `max`, available as a method on `List<Int>`. The maximum of a list should be greater than or equal to every other element in the list. Let's specify this:

Listing 8.8 Property specifying maximum value in a list

```
val smallInt = Gen.choose(-10, 10)

val maxProp = forAll(SGen.listOf(smallInt)) { ns ->
    val mx = ns.max()
        ?: throw IllegalStateException("max on empty list")
    !ns.exists { it > mx }   ❶
}
```

❶ No value greater than `mx` should exist in `ns`

At this point, calling `check` directly on a `Prop` is rather cumbersome. We can introduce a helper function for running property values and printing their result to the console in a useful format. Let's simply call it `run`.

Listing 8.9 A convenience method for running properties using sensible defaults

```
fun run(
    p: Prop,
    maxSize: Int = 100,   ❶
    testCases: Int = 100,   ❷
    rng: RNG = SimpleRNG(System.currentTimeMillis())   ❸
): Unit =
    when (val result = p.check(maxSize, testCases, rng)) {
        is Falsified ->   ❹
            println(
                "Falsified after ${result.successes}" +
                    "passed tests: ${result.failure}"
            )
        is Passed ->   ❺
            println("OK, passed $testCases tests.")
    }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❶ Set default maximum size of test cases to `100`

❷ Set default amount of test cases to run to `100`

❸ Provide a simple random number generator ready for action

❹ Print error message to standard out in case of failure.

❺ Print success message to standard out in case tests pass.

We're taking advantage of some default arguments here, making the method more convenient to call. We want the default number of tests to be enough to get good coverage, yet not too many or they'll take too long to run.

If we try running `run(maxProp)`, we notice that the property fails!

```
Falsified after 0 passed tests: test case: []
generated and exception: max on empty list
stacktrace:
...
```

Property-based testing has a way of revealing hidden assumptions that we have about our code, and forcing us to be more explicit about these assumptions. The standard library's implementation of `max` returns `null` when dealing with empty lists, which we interpreted as an `IllegalStateException`. We need to fix our property to take this into account.

EXERCISE 8.13

Define `nonEmptyListOf` for generating nonempty lists, and then update your specification of `max` to use this generator.

```
fun <A> nonEmptyListOf(ga: Gen<A>): SGen<List<A>> = TODO()

val maxProp = TODO()
```

EXERCISE 8.14

Write a property called `maxProp` to verify the behavior of `List.sorted`, which you can use to sort (among other things) a `List<Int>`.

8.4.2 Writing a test suite for parallel computations

Recall that in chapter 7 we discovered laws that should hold true for our parallel computations. Can we express these laws with our library? The first "law" we looked at was actually a particular test case:

```
map(unit(1)) { it + 1 } == unit(2)
```

We certainly can express this, but the result is somewhat ugly assuming our representation of

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`Par<A>` being an alias for the function type `(ExecutorService) -> Future<A>`.

```
val es = Executors.newCachedThreadPool()
val p1 = forAll(Gen.unit(Pars.unit(1))) { pi ->
    map(pi, { it + 1 })(es).get() == Pars.unit(2)(es).get()
}
```

The resulting test is verbose, cluttered, and the *idea* of the test is obscured by irrelevant detail. Notice that this isn't a question of the API being expressive enough—yes, we can express what we want, but a combination of missing helper functions and poor syntax obscures the real intent.

PROVING PROPERTIES

Next, let's improve on this verbosity and clutter. Our first observation is that `forAll` is a bit too general for this test case. We aren't varying the input to the test, we just have a hardcoded example which in turn should be as convenient to write as in any traditional unit testing framework. Let's introduce a combinator for it on the `Prop` companion object:

```
fun check(p: () -> Boolean): Prop = TODO()
```

How would we implement this? One possible way is to use `forAll`:

```
fun check(p: () -> Boolean): Prop {   ❶
    val result by lazy { p() }   ❷
    return forAll(Gen.unit(Unit)) {
        result
    }
}
```

❶ Pass in a non-strict value

❷ Result is memoized to avoid recomputation

This doesn't seem quite right. We're providing a `unit` generator that only generates a single `Unit` value. Then we proceed by ignoring that value just to force evaluation of the given `Boolean`. Not great.

Even though we memoize the result so that it's not evaluated more than once, the test runner will still generate multiple test cases and test the `Boolean` many times. For example, if we execute `run(check(true))`, this will test the property 100 times and print "OK, passed 100 tests." But checking a property that is always `true` 100 times is a terrible waste of effort. What we need is a new primitive.

The representation of `Prop` that we have so far is just a function of type `(MaxSize, TestCases, RNG) -> Result`, where `Result` is either `Passed` or `Falsified`. A simple implementation of a `check` primitive is to construct a `Prop` that ignores the number of test cases:

```
fun check(p: () -> Boolean): Prop =
    Prop { _, _, _ ->
        if (p()) Passed
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        else Falsified("()", 0)
    }
```

This is certainly better than using `forAll`, but `run(check(true))` will still *print* "passed 100 tests" even though it only tests the property once. It's not really true that such a property has "passed" in the sense that it remains unfalsified after a number of tests. It is *proved* after just one test. It seems that we want a new kind of `Result`:

Listing 8.10 Introduce `Proved` to represent result that has proof after a single test

```
object Proved : Result()
```

We can now return `Proved` instead of `Passed` in a property created by `check`. We'll need to modify the test runner to take this new case into account:

Listing 8.11 Update `run` to handle new `Proved` result type

```
fun run(
    p: Prop,
    maxSize: Int = 100,
    testCases: Int = 100,
    rng: RNG = SimpleRNG(System.currentTimeMillis())
): Unit =
    when (val result = p.run(maxSize, testCases, rng)) {
        is Falsified ->
            println(
                "Falsified after ${result.successes} passed tests: " +
                        result.failure
            )
        is Passed ->
            println("OK, passed $testCases tests.")
        is Proved ->
            println("OK, proved property.")
    }
```

We also need to modify our implementations of `Prop` combinators like `and`. These changes are quite trivial, since such combinators don't need to distinguish between `Passed` and `Proved` results.

Listing 8.12 Update `Prop` to handle both `Passed` and `Proved` passes

```
fun and(p: Prop) =
    Prop { max, n, rng ->
        when (val prop = run(max, n, rng)) {
            is Falsified -> prop
            else -> p.run(max, n, rng)   ❶
        }
    }
```

❶ The `else` fallback handles both `Passed` and `Proved` success types

If you wish to go further, have a look at exercise C.1 in appendix C.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TESTING PAR

Getting back to proving the property that `map(unit(1)) { it + 1 }` is equal to `unit(2)`, we can use our new `check` primitive to express this in a way that doesn't obscure the intent:

```
val p = check {
    val p1 = map(unit(1)) { it + 1 }
    val p2 = unit(2)
    p1(es).get() == p2(es).get()
}
```

This is now pretty clear. But can we do something about the noise of `p1(es).get()` and `p2(es).get()`? This needless repetition obscures the intent of our test and has very little to do with what we are attempting to prove. We're forcing this code to be aware of the internals of `Par` so that we can compare two `Par` values to each other for equality. One improvement is to *lift* the equality comparison into `Par` using `map2`, which means we only have to run a single `Par` at the end to get our result:

```
fun <A> equal(p1: Par<A>, p2: Par<A>): Par<Boolean> =
    map2(p1, p2, { a, b -> a == b })

val p = check {
    val p1 = map(unit(1)) { it + 1 }
    val p2 = unit(2)
    equal(p1, p2)(es).get()
}
```

This is already a bit better than having to run each side separately. But while we're at it, why don't we move the running of `Par` out into a separate function called `forAllPar`. This also gives us a good place to insert variation across different parallel strategies, without it cluttering the property we're specifying:

```
val ges: Gen<ExecutorService> = weighted(   ❶
    Gen.choose(1, 4).map {
        Executors.newFixedThreadPool(it)
    } to .75,   ❷
    Gen.unit(
        Executors.newCachedThreadPool()
    ) to .25)   ❸

fun <A> forAllPar(ga: Gen<A>, f: (A) -> Par<Boolean>): Prop =
    forAll(
        map2(ges, ga) { es, a -> es to a }   ❹
    ) { (es, a) -> f(a)(es).get() }
```

❶ A weighted generator of executor services

❷ Create a fixed thread pool 75% of the time

❸ Create an unbounded thread pool 25% of the time

❹ Create a `Pair<Gen<ExecutorService>, Gen<A>>` using the `to` keyword

The value `ges` is a `Gen<ExecutorService>` that will vary over fixed-size thread pools from 1–4 threads, as well as consider an unbounded thread pool.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Next, let's focus our attention on `map2(ges, ga) { es, a -> es to a }`. This is a rather noisy way of combining two generators to produce a pair of their outputs. Let's introduce a combinator to clean this mess up:

```
fun <A, B> combine(ga: Gen<A>, gb: Gen<B>): Gen<Pair<A, B>> =
    map2(ga, gb) { a, b -> a to b }
```

This already feels a lot better and less clunky!

```
fun <A> forAllPar(ga: Gen<A>, f: (A) -> Par<Boolean>): Prop =
    forAll(
        combine(ges, ga)
    ) { esa ->
        val (es, a) = esa
        f(a)(es).get()
    }
```

Even though this is better, we haven't arrived yet. Our aim is to make the user experience of our library as frictionless as possible. We can make it even easier and more natural to use by applying some features in our Kotlin bag of tricks. For one, we could introduce `combine` as a method on `Gen`. We could also use the `infix` keyword to get rid of unnecessary punctuation and parentheses:

```
infix fun <A, B> Gen<A>.combine(gb: Gen<B>): Gen<Pair<A, B>> =
    map2(this, gb) { s, a -> s to a }
```

Which in turn gives us a far more fluid expression such as this:

```
fun <A> forAllPar(ga: Gen<A>, f: (A) -> Par<Boolean>): Prop =
    forAll(ges combine ga) { esa ->
        val (es, a) = esa
        f(a)(es).get()
    }
```

The final improvement we can make is to improve the injection of parameters into the anonymous function by performing an inline destructure of `Pair<ExecutorService, A>`, bringing us to our final iteration.

```
fun <A> forAllPar(ga: Gen<A>, f: (A) -> Par<Boolean>): Prop =
    forAll(ges combine ga) { (es, a) ->
        f(a)(es).get()
    }
```

We can now go ahead and use our new property to implement `checkPar`, which in turn consumes `Par<Boolean>` as emitted by `Par.equal` from chapter 7. All of this combined makes for a better experience by the users of our library.

```
fun checkPar(p: Par<Boolean>): Prop =
    forAllPar(Gen.unit(Unit)) { p }

val p2 = checkPar(
    equal(
        map(unit(1)) { it + 1 },
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        unit(2)
    )
)
```

With all these stepwise improvements, our property has become easier to understand and use. These might seem like minor changes, but such refactoring and cleanup has a huge effect on the usability of our library. The helper functions we've written make the properties easier to read and more pleasant to work with.

Let's look at some other properties from chapter 7. Recall that we generalized our test case:

```
map(unit(x), f) == unit(f(x))
```

We then simplified it to the law that mapping the identity function over a computation should have no effect:

```
map(y, id) == y
```

Can we express this? Not exactly. This property implicitly states that the equality holds *for all* choices of `y`, for all types. We're forced to pick particular values for `y`:

```
val pint: Gen<Par<Int>> =
    Gen.choose(0, 10).map {
        unit(it)
    }

val p = forAllPar(pint) { n ->
    equal(map(n) { it }, n)
}
```

We can certainly range over more choices of `y`, but what we have here is probably good enough. The implementation of `map` doesn't care about the values of our parallel computation, so there isn't much point in constructing the same test for `Double`, `String`, and so on. What *can* affect `map` is the *structure* of the parallel computation. If we wanted greater assurance that our property held, we could provide richer generators for the structure. Here, we're only supplying `Par` expressions with one level of nesting.

EXERCISE 8.15

Write a richer generator for `Par<Int>` which builds more deeply nested parallel computations than the simple variant we've provided so far.

EXERCISE 8.16

Express the property about `fork` from chapter 7 that `fork(x) == x`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

8.5 Generating higher-order functions and other possibilities

So far, our library seems quite expressive, but there is one area where it's lacking: we don't have a good way to test higher-order functions. While we have lots of ways of generating *data* using our generators, we don't really have a good way of generating *functions*. In this section we deal with generating functions in order to test higher-order functions.

For instance, let's consider the `takeWhile` function defined for `List` and `Sequence`. Recall that this function returns the longest prefix of its input whose elements all satisfy a predicate. For instance, `listOf(1,2,3).takeWhile { it < 3 }` results in `List(1,2)`. A simple property we'd like to check is that for any list, `s: List<A>`, and any `f: (A) -> Boolean`, the expression `s.takeWhile(f).forAll(f)` evaluates to `true`. That is, every element in the returned list satisfies the predicate.[41]

EXERCISE 8.17

Come up with some other properties that `takeWhile` should satisfy. Can you think of a good property expressing the relationship between `takeWhile` and `dropWhile`?

We could certainly take the approach of only examining *particular* arguments when testing higher-order functions. For instance, here's a more specific property for `takeWhile`:

```
val isEven = { i: Int -> i % 2 == 0 }

val takeWhileProp =
    Prop.forAll(Gen.listOfN(n, ga)) { ns ->
        ns.takeWhile(isEven).forAll(isEven)
    }
```

This works, but is there a way we could let the testing framework handle generating functions to use with `takeWhile` instead? Let's consider our options. To make this concrete, let's suppose we have a `Gen<Int>` and would like to produce a `Gen<(String) -> Int>`. What are some ways we could do that? Well, we could produce `(String) -> Int` functions that simply ignore their input string and delegate to the underlying `Gen<Int>`:

```
fun genStringIntFn(g: Gen<Int>): Gen<(String) -> Int> =
    g.map { i -> { _: String -> i } }
```

This approach isn't sufficient though. We're simply generating constant functions that ignore their input. In the case of `takeWhile`, where we need a function that returns a `Boolean`, this will be a function that always returns `true` or `false` depending on what the underlying boolean generator passes it—clearly not very interesting for testing the behavior of our function.

```
fun genIntBooleanFn(g: Gen<Boolean>): Gen<(Int) -> Boolean> =
    g.map { b: Boolean -> { _: Int -> b } }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Now, let's consider the following function that will return a function generator that will perform some logic based on a value passed to it. In this case a threshold `t` is passed, and any `Int` injected into the function will be tested to see if the value exceeds `t`.

```
fun genIntBooleanFn(t: Int): Gen<(Int) -> Boolean> =
    Gen.unit { i: Int -> i > t }
```

Now let's put our new function generator to work. We begin by generating a `List<Int>` as well as a random threshold value. We pre-load the our function generator with the given random threshold and let it produce its function of `(Int) -> Boolean`. Lastly, we apply this generated function to `takeWhile` on our generated list and then apply the same predicate to `forAll` which should always result in `true`.

```
val gen: Gen<Boolean> =
    Gen.listOfN(100, Gen.choose(1, 100)).flatMap { ls: List<Int> ->
        Gen.choose(1, ls.size / 2).flatMap { threshold: Int ->
            genIntBooleanFn(threshold).map { fn: (Int) -> Boolean ->
                ls.takeWhile(fn).forAll(fn)
            }
        }
    }
```

When run in context of our test harness using `Prop.forAll`, we should always see the test passing.

```
run(Prop.forAll(gen) { success -> success })
```

Even though this example is somewhat contrived and trivial, it sufficiently demonstrates what is possible in terms of random function generators. We can take these ideas much further, and suggest that you explore the options by following the expansive exercises found in appendix C at the back of the book.

8.6 The laws of generators

As we've worked through the process of designing our library, we see patterns emerging that we've come across many times before in previous chapters. Many of the combinators we've discovered even have the same name and functionality. For example, many of the functions we've implemented for our `Gen` type look quite similar to other functions we defined on `Par`, `List`, `Stream`, and `Option`. Looking back at our implementation of `Par` in chapter 7 reveals that we defined the following combinator:

```
fun <A, B> map(a: Par<A>, f: (A) -> B): Par<B> = TODO()
```

And in this chapter we defined `map` for `Gen` (as a method on `Gen<A>`):

```
fun <A, B> map(a: Gen<A>, f: (A) -> B): Gen<B> = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

We've also defined similar-looking functions for `Option`, `List`, `Stream`, and `State`. We have to wonder, is it merely that our functions share similar-looking signatures, or do they satisfy the same *laws* as well? Let's look at a law we introduced for `Par` in chapter 7:

```
map(y, id) == y
```

Does this law hold true for our implementation of `Gen.map`? What about for `Stream`, `List`, `Option`, and `State`? Yes, it does! Try it and see. This indicates that not only do these functions share similar-looking signatures, they also in some sense have analogous meanings in their respective domains. It appears there are deeper forces at work! We're uncovering some fundamental patterns that cut across all these domains. In part 3, we'll learn the names for these patterns, discover the laws that govern them, and understand what it all means.

To reiterate, the goal was not necessarily to learn about property-based testing as such, but rather to highlight particular aspects of functional design. First, we saw that oscillating between the abstract algebra and the concrete representation lets the two inform each other. This avoids over-fitting the library to a particular representation, and also avoids a disconnected abstraction which is far removed from the end goal.

Second, we noticed that this domain led us to discover many of the same combinators we've now seen a few times before: `map`, `flatMap`, and so on. Not only are the signatures of these functions analogous, the *laws* satisfied by the implementations are analogous too. There are a great many seemingly distinct *problems* being solved in the world of software, yet the space of functional *solutions* is much smaller. Many libraries are just simple combinations of certain fundamental structures that appear over and over again across a variety of different domains. This is an opportunity for code reuse that we'll exploit in part 3. We will learn both the names of these structures as well as how to spot more general abstractions.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

8.7 Summary

- You can use property-based testing to validate laws or properties that relate functions to each other.
- Building a property-based testing library is an excellent example of how to design a functional library using an iterative approach.
- You can model a simple testing library using data types representing properties and generators to affirm the laws of your program.
- You can use generators with other generators to express complex laws when validating the code under test.
- It is possible to minimize test case output by shrinking applied test data or using incremental sized generation.
- The user experience of a library is very important, and usability should always be a primary goal of library design.
- You can design functional libraries using an oscillation between abstract algebra and concrete representation.
- Combinators across domains obey the same laws and have the same semantics, which establish universal functional design patterns.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

9 Parser combinators

This chapter covers:

- Applying an algebraic design approach to libraries
- Starting with simple examples and progressively introducing complexity during design
- Drawing distinction between primitives and higher level combinators
- Inventing and adapting combinators to achieve design goals
- Improving library ergonomics by introducing syntactic sugar
- Postponing combinator implementation by first focusing on algebra design

In this chapter, we'll work through the design of a combinator library for creating *parsers*. We'll use JSON[42] parsing as a motivating use case. Like chapters 7 and 8, this chapter is not so much about parsing as it is about providing further insight into the process of functional design.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **What is a parser?**

A parser is a specialized program that takes unstructured data (such as text, or any kind of stream of symbols, numbers, or tokens) as input, and outputs a structured representation of that data. For example, we can write a parser to turn a comma-separated file into a list of lists, where the elements of the outer list represent the records, and the elements of each inner list represent the comma-separated fields of each record. Another example is a parser that takes an XML or JSON document and turns it into a tree-like data structure.

In a parser combinator library like the one we'll build in this chapter, a parser doesn't have to be anything quite that complicated, and it doesn't have to parse entire documents. It can do something as elementary as recognizing a single character in the input. We then use combinators to assemble composite parsers from elementary ones, and still more complex parsers from those.

This chapter will introduce a design approach that we'll call *algebraic design*. This design approach is just a natural evolution of what we've already been doing to different degrees in past chapters—designing our interface first, along with associated laws, and letting the combination of these guide our choice of data type representations.

At a few key points during this chapter, we'll give more open-ended exercises, intended to mimic the scenarios you might encounter when writing your own libraries from scratch. You'll get the most out of this chapter if you use these opportunities to put the book down and spend some time investigating possible approaches. When you design your own libraries, you won't be handed a neatly chosen sequence of type signatures to fill in with implementations. Instead you'll have to make the decisions about what types and combinators you need, and a goal of this part of the book has been to prepare you for doing this on your own. As always, if you get stuck on one of the exercises or want some more ideas, you can keep reading or consult the answers in Appendix B. It may also be a good idea to do these exercises with another person, or even compare notes with other readers of the liveBook edition.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Parser combinators versus parser generators**

You might be familiar with *parser generator* libraries like Yacc[43] or similar libraries in other languages (for instance, ANTLR[44] in Java). These libraries generate code for a parser based on a specification of the grammar. This approach works fine and can be quite efficient, but comes with all the usual problems of code generation—the libraries produce as their output a monolithic chunk of code that's difficult to debug. It's also difficult to reuse fragments of logic, since we can't introduce new combinators or helper functions to abstract over common patterns in our parsers.

In a parser combinator library, parsers are just ordinary first-class values. Reusing parsing logic is trivial, and we don't need any sort of external tool separate from our programming language.

9.1 Designing an algebra

Recall from section 7.4 that we defined *algebra* to mean a collection of functions operating over some data types, *along with a set of laws* specifying relationships between these functions. In past chapters, we moved rather fluidly between inventing functions in our algebra, refining the set of functions, and tweaking our data type representations. Laws were somewhat of an afterthought—we worked out the laws only after we had a representation and an API fleshed out. There's nothing wrong with this style of design, but in this chapter we'll take a different approach. We'll *start* with the algebra (including its laws) and decide on a representation later. This approach—let's call it *algebraic design* —can be used for any design problem, but works particularly well for parsing. This is mostly because it's easy to imagine what combinators are required for parsing different kinds of inputs. This in turn lets us keep an eye on the concrete goal even as we defer deciding on a representation.

There are many different kinds of parsing libraries. There are even several open source Kotlin parser combinator libraries available. As in the previous chapter, we're deriving our own library from first principles partially for pedagogical purposes, and to further encourage the idea that no library is authoritative. Ours will be designed for expressiveness, by being able to parse arbitrary grammars, as well as for speed, and good error reporting. This last point is important. Whenever we run a parser on input that isn't expected—which can happen if the input is malformed—it should generate a parse error. If there are parse errors, we want to be able to point out exactly where the error is in the input and accurately indicate its cause. Error reporting is often an afterthought in parsing libraries, but we'll make sure that we give careful attention to it from the start.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

9.1.1 A parser to recognize single characters

Okay, let's begin. For simplicity and for speed, our library will create parsers that operate on strings as input. We could make the parsing library more generic, but we will refrain as this will come at some cost. We need to pick some parsing tasks to help us discover a good algebra for our parsers. What should we look at first? Something practical like parsing an email address, JSON, or HTML? No! These tasks can come later. A good and simple domain to start with is parsing various combinations of repeated letters and gibberish words like `"abracadabra"` and `"abba"`. As silly as this sounds, we've seen before how simple examples like this help us ignore extraneous details and focus on the essence of the problem.

So let's start with the simplest of parsers, one that recognizes the single character input `'a'`. As in past chapters, we can just *invent* a combinator for the task and call it `char`:

```
fun char(c: Char): Parser<Char>
```

What have we done here? We've conjured up a type called `Parser` which is parameterized on a single parameter indicating the *result type* of the `Parser`. That is, running a parser shouldn't simply yield a yes/no response—if it succeeds, we want to get a *result* that has some useful type, and if it fails, we expect *information about the failure*. The `char('a')` parser will succeed only if the input is exactly the character `'a'` and it will return that same character `'a'` as its result.

This talk of "running a parser" makes it clear that our algebra needs to be extended somehow to support that. Let's invent another function for it:

```
fun <A> run(p: Parser<A>, input: String): Either<PE, A>
```

Wait a minute, what does `PE` represent? It's a type parameter we just conjured into existence! At this point, we don't care too much about the representation of `PE` (short for parse error), or `Parser` for that matter. We're in the process of specifying an *interface* that happens to make use of two types whose representation or implementation details we choose to remain ignorant of as much as possible. Let's make this explicit with some interface declarations:

Listing 9.1 An interface that provides a place to declare all Parser combinators.

```
interface Parsers<PE> {   ❶

    interface Parser<A>   ❷

    fun char(c: Char): Parser<Char>

    fun <A> run(p: Parser<A>, input: String): Either<PE, A>

}
```

❶ Interface parameterized with parse error `PE`, where all future parser combinators may be declared.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❷ A simple representation of the parser.

In listing 9.1, a top level interface called `Parsers` is introduced. This will become the home for all combinators and helper functions relating to the `Parser` and related `PE` parser error. For now, we keep both these types in their simplest representation and will keep adding new combinators to the body of the `Parsers` interface.

Placing our attention back on the `char` function, we should satisfy an obvious law—for any `c` of type `Char`,

```
run(char(c), c.toString()) == Right(c)
```

9.1.2 A parser to recognize entire strings

Let's continue. We can recognize the single character `'a'`, but what if we want to recognize the string `"abracadabra"`? We don't have a way of recognizing entire strings yet, so let's add a function to `Parsers` that helps us with constructing a `Parser<String>`.

```
fun string(s: String): Parser<String>
```

Likewise, this should satisfy an obvious law—for any `String`, `s`,

```
run(string(s), s) == Right(s)
```

What if we want to recognize either string `"abra"` or `"cadabra"`? We could add a very specialized combinator for this purpose:

```
fun orString(s1: String, s2: String): Parser<String>
```

But choosing between two parsers seems like something that would be more useful in a *general* way regardless of their result type. Let's go ahead and make this polymorphic:

```
fun <A> or(pa: Parser<A>, pb: Parser<A>): Parser<A>
```

We expect that `or(string("abra"), string("cadabra"))` will succeed whenever either `string` parser succeeds:

```
run(or(string("abra"), string("cadabra")), "abra") ==
    Right("abra")
run(or(string("abra"), string("cadabra")), "cadabra") ==
    Right("cadabra")
```

Even though this works, it is difficult to understand for the reader. Let's do some work on our presentation. We can give this `or` combinator friendlier *infix* syntax where we omit all `.` and parentheses, like `s1 or s2`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.2 Adding syntactic sugar to make the `or` combinator easier to use

```
interface Parsers<PE> {

    interface Parser<A>

    fun string(s: String): Parser<String>   ❶

    fun <A> or(a1: Parser<A>, a2: Parser<A>): Parser<A>   ❷

    infix fun String.or(other: String): Parser<String> =
        or(string(this), string(other))   ❸

    fun <A> run(p: Parser<A>, input: String): Either<PE, A>
}
```

❶ The string parser for turning `String` into `Parser<String>`

❷ The `or` combinator for deciding between two instances of `Parser<A>`

❸ Infix extension method to make `or` combinator more pleasing to use on `String`s

We introduce a convenient `or` extension method on `String` that is marked with the `infix` modifier. The method will lift two adjoining `String`s into `Parser<String>` instances, then apply the `or` combinator on both parsers. This now allows us to declare the law for `or` as follows.

```
run("abra" or "cadabra", "abra") == Right("abra")
```

9.1.3 A parser to recognize repetition

Much neater! We can now recognize various strings, but we don't have a way of talking about repetition. For instance, how would we recognize three repetitions of our `"abra" or "cadabra"` parser? Once again, let's add a combinator to serve this purpose. This *should* remind you of a similar function that we wrote in the previous chapter on property based testing.

```
fun <A> listOfN(n: Int, p: Parser<A>): Parser<List<A>>
```

We made `listOfN` parametric in the choice of `A`, since it doesn't seem like it should care whether we have a `Parser<String>`, a `Parser<Char>`, or some other type of parser. Here are some examples of what we expect from `listOfN` expressed through laws:

```
run(listOfN(3, "ab" or "cad"), "ababab") == Right("ababab")
run(listOfN(3, "ab" or "cad"), "cadcadcad") == Right("cadcadcad")
run(listOfN(3, "ab" or "cad"), "ababcad") == Right("ababcad")
run(listOfN(3, "ab" or "cad"), "cadabab") == Right("cadabab")
```

At this point, we've just been accumulating required combinators, but we haven't tried to refine our algebra into a minimal set of primitives. We also haven't talked much about more general laws. We'll start doing this next, but rather than give the game away, we'll ask you to examine a few more simple use cases yourself while trying to design a minimal algebra with associated laws. This should be a challenging task, but enjoy wrestling with it and see what you can come up with.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Here are additional parsing tasks to consider, along with some guiding questions:

- A `Parser<Int>` that recognizes zero or more `'a'` characters, and whose result value is the number of `'a'` characters it has seen. For instance, given `"aa"`, the parser results in `2;` given `""` or `"b123"` (a string not starting with `'a'`), it results in `0;` and so on.
- A `Parser<Int>` that recognizes *one* or more `'a'` characters, and whose result value is the number of 'a' characters it has seen. Is this defined somehow in terms of the same combinators as the parser for `'a'` repeated zero or more times? The parser should fail when given a string without a starting 'a'. How would you like to handle error reporting in this case? Could the API support giving an explicit message like `"Expected one or more 'a'"` in the case of failure?
- A parser that recognizes zero or more `'a'`, followed by one or more `'b'`, and which results in the pair of counts of characters seen. For instance, given `"bbb"`, we get `Pair(0,3)`, given `"aaaab"`, we get `Pair(4,1)`, and so on.

Some additional considerations:

- If we're trying to parse a sequence of zero or more `"a"` and are only interested in the number of characters seen, it seems inefficient to have to build up a `List<Char>`, only to throw it away and extract the length. Could something be done about this?
- Are the various forms of repetition primitive in our algebra, or could they be defined in terms of something simpler?
- We introduced a type parameter `PE` representing parse errors earlier, but so far we haven't chosen any representation or functions for its API. Our algebra also doesn't have any way of letting the programmer control what errors are reported. This seems like a limitation, given that we'd like meaningful error messages from our parsers. Can something be done about this?
- Does `a or (b or c)` mean the same thing as `(a or b) or c`? If yes, is this a primitive law for your algebra, or is it implied by something simpler?
- Try to come up with a set of laws to specify your algebra. You don't necessarily need the laws to be complete, just write down some laws that you expect should hold for any `Parsers` implementation.

Spend some time coming up with combinators and possible laws based on this guidance. When you feel stuck or at a good stopping point, then continue by reading the next section, which walks through one possible algebra design to meet these requirements.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **The advantages of algebraic design**

When you design the algebra of a library *first*, representations for the data types of the algebra don't matter as much. As long as they support the required laws and functions, you don't even need to make your representations public.

There's an underlying idea here that a type is given meaning based on its relationship to other types (which are specified by the set of functions and their laws), rather than its internal representation. This viewpoint is often associated with category theory, a branch of mathematics we've alluded to before.

9.2 One possible approach to designing an algebra

In this section, we'll walk through the discovery process of a set of combinators for the parsing tasks mentioned earlier. If you worked through this design task yourself, you likely took a different path to what we will take. You may well have ended up with a different set of combinators, which is perfectly fine.

9.2.1 Counting character repetition

First, let's consider the parser that recognizes zero or more repetitions of the character `'a'` and returns the number of characters it has seen. We can start by adding a primitive combinator that takes us halfway there, let's call it `many`:

```
fun <A> many(pa: Parser<A>): Parser<List<A>>
```

This isn't exactly what we're after—we need a `Parser<Int>` that counts the number of elements. We could change the `many` combinator to return a `Parser<Int>`, but that feels too specific. Undoubtedly there will be occasions where we care about more than just the list length. Better to introduce another combinator that *should* be familiar by now, `map`:

```
fun <A, B> map(pa: Parser<A>, f: (A) -> B): Parser<B>
```

We can now define our parser as follows:

```
map(many(char('a'))) { it.size }
```

Let's transform these combinators into extension methods to make this a bit more pleasing to the eye.

```
   fun <A> Parser<A>.many(): Parser<List<A>>

   fun <A, B> Parser<A>.map(f: (A) -> B): Parser<B>
```

With these combinators in place, our new parser may be expressed as `numA`, followed by its

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

proof:

```
val numA: Parser<Int> = char('a').many().map { it.size }

run(numA, "aaa") == Right(3)
run(numA, "b") == Right(0)
```

When passing a string comprised of `"aaa"` we expect a count of 3. Similarly, if passed a string of `"b"` we expect a count of 0.

We have a strong expectation for the behavior of `map`. It should merely transform the result value if the `Parser` was successful, and no additional input characters should be examined by `map`. Also, a failing parser can't become a successful one via `map` or vice versa. In general, we expect `map` to be *structure preserving*, much like we required for `Par` and `Gen`. Let's formalize this by stipulating the now-familiar law:

```
map(p) { a -> a } == p
```

How should we document this law? We could put it in a documentation comment, but in the preceding chapter we developed a way to make our laws *executable*. Let's use our property-based testing library here!

```
object ParseError   ❶

abstract class Laws : Parsers<ParseError> {   ❷
    private fun <A> equal(   ❸
        p1: Parser<A>,
        p2: Parser<A>,
        i: Gen<String>
    ): Prop =
        forAll(i) { s -> run(p1, s) == run(p2, s) }

    fun <A> mapLaw(p: Parser<A>, i: Gen<String>): Prop =   ❹
        equal(p, p.map { a -> a }, i)
}
```

❶ Concrete implementation `ParseError` for type parameter `PE` in `Parsers`

❷ Implement `Parsers` interface allowing access to all combinators and helper functions.

❸ A helper function for asserting parser equality

❹ A property that tests if our `map` function obeys the law

The `Laws` class is declared as abstract for now until we've implemented all methods in the `Parsers` interface, at which point this can become an object. We now have a way to test if our combinator holds true for the specified law. This will come in handy later when we test that our implementation of `Parsers` behaves as we expect. When we discover more laws later on, you are encouraged to write them out as actual properties inside the `Laws` class. For the sake of brevity, we won't give `Prop` implementations of all the laws, but that doesn't mean you shouldn't write them yourself!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Incidentally, if we consider `string` to be one of our core primitive functions, combined with `map` we could easily implement `char` in terms of `string`:

```
fun char(c: Char): Parser<Char> = string(c.toString()).map { it[0] }
```

And similarly, another combinator called `succeed` can be defined in terms of `string` and `map`. This parser always succeeds with the value `a`, regardless of the input string (since `string("")` will always succeed, even if the input is empty):

```
fun <A> succeed(a: A): Parser<A> = string("").map { a }
```

Does this combinator seem familiar to you? We can specify its behavior with a law:

```
run(succeed(a), s) == Right(a)
```

9.2.2 Slicing and nonempty repetition

The combination of `many` and `map` certainly lets us express the parsing task of counting the number of `'a'` characters that we have seen, but it seems inefficient to construct a `List<Char>` only to discard its values and extract its length. It would be nice if we could run a `Parser` purely to see what portion of the input string it examines. Let's come up with a combinator for that very purpose, call it `slice`:

```
fun <A> slice(pa: Parser<A>): Parser<String>
```

We call this combinator `slice` since we intend for it to return the portion of the input string examined by the parser if successful. As an example:

```
run(slice(('a' or 'b').many()), "aaba") == Right("aaba")
```

We ignore the list accumulated by `many` and simply return the portion of the input string matched by the parser. With `slice` converted to an extension method, our parser that counts `'a'` characters can now be written as:

```
char('a').many().slice().map { it.length }
```

The `length` field refers to `String.length`, which takes constant time. This is different to the `size()` method on `List` which may take time proportional to the length of the list[45], and subsequently requires us to construct the list before we can count its elements.

Note that there is no implementation yet. We're merely coming up with our desired interface. But `slice` does put a constraint on the implementation: even if the parser: `p.many().map { it.size() }` will generate an intermediate list when run, `p.many().slice().map { it.length }` will not. This is a strong hint that `slice` is primitive, since it will have to have access to the internal representation of the parser.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Let's consider the next use case. What if we want to recognize *one or more* `'a'` characters? First, we introduce a new combinator for it, `many1`:

```
fun <A> many1(p: Parser<A>): Parser<List<A>>
```

It feels like `many1` shouldn't have to be primitive, but must be defined in terms of `many`. In actual fact, `many1(p)` is just `p` *followed by* `many(p)`. So it seems we need some way of running one parser, followed by another, assuming the first is successful. Let's accommodate running parsers sequentially by adding a `product` combinator:

```
fun <A, B> product(pa: Parser<A>, pb: Parser<B>): Parser<Pair<A, B>>
```

We can now add an infix product extension method to `Parser<A>` that allows us to express `pa product pb`:

```
infix fun <A, B> Parser<A>.product(
    pb: Parser<B>
): Parser<Pair<A, B>>
```

Up to this point there has been a complete focus on driving development from algebra alone. We will continue to keep this approach, but let's have some fun and implement some combinators!

EXERCISE 9.1

Using `product`, implement the now-familiar combinator `map2`. In turn, use this to implement `many1` in terms of `many`.

```
override fun <A, B, C> map2(
    pa: Parser<A>,
    pb: () -> Parser<B>,
    f: (A, B) -> C
): Parser<C> = TODO()

override fun <A> many1(p: Parser<A>): Parser<List<A>> = TODO()
```

With `many1`, we can now implement the parser for zero or more `'a'` followed by one or more `'b'` as follows:

```
char('a').many().slice().map { it.length } product
    char('b').many1().slice().map { it.length }
```

EXERCISE 9.2 (Hard)

Try coming up with laws to specify the behavior of `product`.

Now that we have `map2`, is `many` really primitive? Let's think about what `many(p)` will do. It will try running `p`, *followed by* `many(p)` again, and again, and so on until the attempt to parse `p` fails. It'll accumulate the results of all successful runs of `p` into a list. As soon as `p` fails, the parser

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

returns the empty `List`.

EXERCISE 9.3 (Hard)

Before continuing, see if you can define `many` in terms of `or`, `map2`, and `succeed`.

```
fun <A> many(pa: Parser<A>): Parser<List<A>> = TODO()
```

EXERCISE 9.4 (Hard)

Implement the `listOfN` combinator introduced earlier using `map2` and `succeed`.

```
fun <A> listOfN(n: Int, pa: Parser<A>): Parser<List<A>> = TODO()
```

We've already had a stab at implementing `many` in exercise 9.3. Let's try to work through this problem together to see what we can learn. Here's the implementation in terms of `or`, `map2`, and `succeed`:

```
infix fun <T> T.cons(la: List<T>): List<T> = listOf(this) + la

fun <A> many(pa: Parser<A>): Parser<List<A>> =
    map2(pa, many(pa)) { a, la ->
        a cons la
    } or succeed(emptyList())
```

We start by adding a neat little extension method that allows for creating a list by prefixing an element to a list of elements, we call it `cons`. This is merely some syntactic sugar that replaces `listOf(a) + la` by `a cons la`, and makes the code a bit easier to comprehend.

The implementation of `many` looks tidy and declarative. We're using `map2` to express that we want `p` followed by `many(p)`, and that we want to combine their results with `cons` to construct a list of results. Or, if that fails, we want to `succeed` with an empty list. But there's a problem with this implementation. We're calling `many` recursively in the second argument to `map2`, which is a *strict* evaluation of its second argument. Consider a simplified program trace of the evaluation of `many(p)` for some parser `p`. We're only showing the expansion of the left side of the `or` here:

```
many(p)
map2(p, many(p)) { a, la -> a cons la }
map2(p, map2(p, many(p)) { a, la -> a cons la }) { a, la ->
    a cons la
}
```

Because a call to `map2` always evaluates its second argument, our `many` function will never terminate! That's no good. This indicates that we need to make `product` and `map2` non-strict in their second arguments:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A, B> product(
    pa: Parser<A>,
    pb: () -> Parser<B>
): Parser<Pair<A, B>> = TODO()

fun <A, B, C> map2(
    pa: Parser<A>,
    pb: () -> Parser<B>,
    f: (A, B) -> C
): Parser<C> =
    product(pa, pb).map { (a, b) -> f(a, b) }
```

EXERCISE 9.5

We could also deal with non-strictness using a separate combinator like we did in chapter 7. Provide an new combinator called `defer` and make the necessary changes to your existing combinators. What do you think of that approach in this instance?

NOTE The purpose of this exercise is merely to try the approach of introducing a `defer` function, and the impact that it would have on our existing combinators. We won't be introducing it beyond this exercise because of the complexity it adds, along with limited benefit that it gives our library. That said, it was worth having a play and trying it out!

By updating our implementation of `many` to take advantage of the second lazy parameter of `map2` by using `defer`, our problem now goes away:

Listing 9.3 Implementation of `many` that relies on lazy evaluation

```
fun <A> many(pa: Parser<A>): Parser<List<A>> =
    map2(pa, many(pa).defer()) { a, la ->   ❶
        a cons la
    } or succeed(emptyList())
```

❶ Second parameter to `map2` becomes a thunk.

Because `map2` draws on the functionality of `product`, it should be non-strict in its second argument too since if the first `Parser` fails, the second won't even be consulted.

We now have good combinators for parsing one thing followed by another, or multiple things of the same kind in succession. But since we're considering whether combinators should be non-strict, let's revisit the `or` combinator from earlier:

```
fun <A> or(pa: Parser<A>, pb: Parser<A>): Parser<A>
```

We'll assume that `or` is left-biased, meaning it tries `p1` on the input and then tries `p2` only if `p1` fails. This is purely a design choice, and you may prefer to have a version of `or` that always runs both `p1` *and* `p2`. In our case we opt for the non-strict version with the second argument which may never even be consulted. This is what such an `or` combinator would look like:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> or(pa: Parser<A>, pb: () -> Parser<A>): Parser<A>
```

9.3 Handling context sensitivity

In this section we will explore a combinator that allows us to pass context on to the next combinator. We call this ability for a combinator to pass state *context sensitivity*.

Let's pause now for a few seconds and reflect on what we've covered so far in this chapter. We've already come a long way in defining a set of useful primitives that we can use in subsequent sections. Here's a review of the most useful ones that we've defined.

Table 9.1 A list of useful primitives derived so far

Primitive	Description
`string(s)`	Recognizes and returns a single `String`
`slice(p)`	Returns the portion of input inspected by `p` if successful
`succeed(a)`	Always succeeds with the value `a`
`map(p, f)`	Applies the function `f` to the result of `p` if successful
`product(p1, p2)`	Sequences two parsers, running `p1` and then `p2`, then returns the pair of their results if both succeed
`or(p1, p2)`	Chooses between two parsers, first attempting `p1`, then passing `p2` any uncommitted input in case `p1` failed

Using these primitives we can express various forms of repetition (`many`, `listOfN`, and `many1`) as well as combinators like `char` and `map2`. Would it surprise you if these primitives were sufficient for parsing *any* context-free grammar, including JSON? Well, they are! We'll get to writing that JSON parser soon, but we need a few more building blocks to get there.

Suppose we want to parse a single digit like `'4'`, followed by *as many* `'a'` characters as that digit. Examples of this kind of input are `"0"`, `"1a"`, `"2aa"`, `"4aaaa"` and so on. This is an example of a *context-sensitive grammar*, and can't be expressed with the `product` primitive that we've defined already. The reason being that the choice of the second parser depends on the result of the first. In other words, the second parser depends on the *context* of the first. Back to our example, we want to run the first parser to extract the digit, then do a `listOfN` using the number from the first parser's result. The `product` combinator simply can't express something like this.

This progression might seem familiar to you. In past chapters we encountered similar situations and dealt with this by introducing a new primitive called `flatMap`.

```
fun <A, B> flatMap(pa: Parser<A>, f: (A) -> Parser<B>): Parser<B>
```

Can you see how this combinator solves the problem of context sensitivity as mentioned above? It provides an ability to sequence parsers, where each parser in the chain depends on the output of the previous one.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 9.6

Using `flatMap` and any other combinators, write the context-sensitive parser we couldn't express earlier. The result should be a `Parser<Int>` that returns the number of characters read. You can make use of a new primitive called `regex` to parse digits, which promotes a regular expression `String` to a `Parser<String>`.

EXERCISE 9.7

Implement `product` and `map2` in terms of `flatMap` and `map`.

EXERCISE 9.8

`map` is no longer primitive. Express it in terms of `flatMap` and/or other combinators.

We have now introduced a new primitive called `flatMap` which enables context-sensitive parsing and allows us to implement `map` and `map2`. This is not the first time `flatMap` has come to the rescue.

Our list of primitives has now shrunk down to six: `string`, `regex`, `slice`, `succeed`, `or`, and `flatMap`. Even though we have less primitives, we have more capability than before by adopting the more general `flatMap` in favor of `map` and `product`. This new power tool enables us to parse arbitrary context-free grammars like JSON, as well as context-sensitive grammars including extremely complex ones like C++ and PERL!

NOTE Up to now, we have spent very little time implementing any of these primitives, and have rather worked on fleshing out the *algebra* by defining abstract definitions in our `Parsers` interface. Let's persist with this approach and defer implementation of these primitives as much as possible.

9.4 Writing a JSON parser

Up to this point we have been building up a set of primitives that give us the basic building blocks to construct more complex parsers. We have managed to parse characters, strings, recognize repetitions and pass context. In this section we will build upon the list of primitives that have been derived so far by developing something of real use, a JSON parser. This is the fun part, so let's jump right into it!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

9.4.1 Defining expectations of a JSON parser

We don't have an implementation of our algebra yet, nor do we have combinators for good error reporting. Our JSON parser doesn't need to know the internal details of how parsers are represented, so we can deal with this later. We can simply write a function that produces a JSON parser using only the set of primitives we've defined, as well as any derived combinators we may need along the way.

We will review the JSON format in a minute, but let's first examine the parse result type that we will expect as an outcome of building our parser. The final outcome will be a structure that looks *something* like this:

Listing 9.4 Top level constructs to be developed for JSON parsing

```
object JSONParser : ParsersImpl<ParseError>() {   ❶
    val jsonParser: Parser<JSON> = TODO()   ❶
}
```

❶ Give access to algebra implementations

❷ Top level declaration for `Parser<JSON>`, with `JSON` to be defined shortly.

Defining this top level function at such an early stage might seem like a peculiar thing to do, since we won't actually be able to run our parser until we have a concrete implementation of the `Parsers` interface. But we'll proceed anyway since it's common FP practice to define an algebra and explore its expressiveness *prior* to defining an implementation. A concrete implementation can tie us down and makes changes to the API more difficult. This is especially true during the design phase of a library. It is much easier to refine an algebra *without* having to commit to any particular implementation. This algebra-first design approach is radically different to what has been taken so far in this book, but is probably the most important lesson to be learned in this chapter.

In section 9.5 we'll return to the question of adding better error reporting to our parsing API. We can do this without disturbing the overall structure of the API or changing the JSON parser very much. We'll also come up with a concrete, runnable representation of our `Parser` type. Importantly, the JSON parser we'll implement in this next section will be completely independent of that representation.

9.4.2 Reviewing the JSON format

If you aren't already familiar with the JSON format, this section will briefly introduce the main concepts of this data representation. You may also want to read Wikipedia's description[46] and the official grammar specification[47] if you want to know more. Here's an example of a simple JSON document:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.5 An example JSON object that can be parsed

```
{
  "Company name" : "Microsoft Corporation",
  "Ticker": "MSFT",
  "Active": true,
  "Price": 30.66,
  "Shares outstanding": 8.38e9,
  "Related companies": [ "HPQ", "IBM", "YHOO", "DELL", "GOOG" ]
}
```

A *value* in JSON can be one of several types. An *object* in JSON is a comma-separated sequence of key-value pairs wrapped in curly braces (`{}`). The keys must be strings like `"Ticker"` or `"Price"`, and the values can be either another object, an *array* like `["HPQ", "IBM" … ]` that contains further values, or a *literal* like `"MSFT", true, null,` or `30.66`.

We'll write a rather dumb parser that simply parses a syntax tree from the document without doing any further processing. Next, we'll need a representation for a parsed JSON document. Let's introduce a data type for this purpose:

Listing 9.6 A data type representing a JSON object to be used for parsing

```
sealed class JSON {
    object JNull : JSON()
    data class JNumber(val get: Double) : JSON()
    data class JString(val get: String) : JSON()
    data class JBoolean(val get: Boolean) : JSON()
    data class JArray(val get: List<JSON>) : JSON()
    data class JObject(val get: Map<String, JSON>) : JSON()
}
```

9.4.3 A JSON parser

The primitives we have developed so far aren't very useful by themselves, but when used as building blocks for something bigger, they suddenly have much more value. Let's review our current list of primitives.

Table 9.2 Primitives to be used as basis for JSON parsing combinators

Primitive	Description
`string(s)`	Recognizes and returns a single `String`
`regex(p)`	Recognizes a regular expression of `String`
`slice(p)`	Returns the portion of input inspected by `p` if successful
`succeed(a)`	Always succeeds with the value `a`
`flatMap(p, f)`	Runs a parser, then uses its result to select a second parser to run in sequence
`or(p1, p2)`	Chooses between two parsers, first attempting `p1`, then `p2` if `p1` fails

In addition, we have used these primitives to define a number of combinators like `map`, `map2`, `many`, and `many1`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 9.9 (Hard)

At this point, you are going to take over the design process. You'll be creating a `Parser<JSON>` from scratch using the primitives we've defined. You don't need to worry about the representation of `Parser` just yet. As you go, you'll undoubtedly discover additional combinators and idioms, notice and factor out common patterns, and so on. Use the skills you've been developing throughout this book, and have fun! If you get stuck, you can always consult the tips in appendix A or the final solution in appendix B.

Here are some basic guidelines to help you in the exercise:

- Any general-purpose combinators you discover can be declared in the `Parsers` abstract class directly. These are top level declarations with no implementation.
- Any syntactic sugar can be placed in another abstract class called `ParsersDsl` that extends from `Parsers`. Make generous use of `infix`, along with anything else in your Kotlin bag-of-tricks to make the final `JSONParser` as easy to use as possible. The functions implemented here should all delegate to declarations in `Parsers`.
- Any JSON-specific combinators can be added to `JSONParser` which extends `ParsersDsl`.
- You'll probably want to introduce combinators that make it easier to parse the tokens of the JSON format (like string literals and numbers). For this you could use the `regex` primitive we introduced earlier. You could also add a few primitives like `letter`, `digit`, `whitespace`, and so on, for building up your token parsers.

NOTE This exercise is about defining the algebra consisting of primitive and combinator declarations only. No implementations should appear in the final solution.

The basic skeleton of what you will be building should resemble something like this:

```
abstract class Parsers<PE> {

    // primitives

    internal abstract fun string(s: String): Parser<String>

    internal abstract fun regex(r: String): Parser<String>

    internal abstract fun <A> slice(p: Parser<A>): Parser<String>

    internal abstract fun <A> succeed(a: A): Parser<A>

    internal abstract fun <A, B> flatMap(
        p1: Parser<A>,
        f: (A) -> Parser<B>
    ): Parser<B>

    internal abstract fun <A> or(
        p1: Parser<out A>,
        p2: () -> Parser<out A>
    ): Parser<A>
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    // other combinators here
}

abstract class ParsersDsl<PE> : Parsers<PE>() {
    // syntactic sugar here
}

abstract class JSONParsers : ParsersDsl<ParseError>() {
    val jsonParser: Parser<JSON> = TODO()
}
```

Take a deep breath and have lots of fun!

9.5 Surfacing errors through reporting

So far we haven't discussed error reporting at all. We've focused exclusively on discovering a set of primitives that allow to us express parsers for different grammars. Aside from parsing a grammar, we also want our parser to respond in a meaningful way when given unexpected input.

Even without knowing what an implementation of `Parsers` will look like, we can reason abstractly about what information is being specified by a set of combinators. None of the combinators we've introduced so far say anything about *what error message* should be reported in the event of failure or what other information a `ParseError` should contain. Our existing combinators only specify what the grammar is and what to do with the result if successful. If we were to declare ourselves done with design, moving us onto implementation of the primitives and combinators, we'd have to make some arbitrary decisions about error reporting and error messages that are unlikely to be universally appropriate.

In this section we will begin to discover a set of combinators for expressing what errors get reported by a `Parser`. Before we dive in, let's think about some pointers to consider during our discovery process.

- Given the following parser:

```
val spaces = string(" ").many()

string("abra") product spaces product string("cadabra")
```

What sort of error would you like to report given the input `"abra cAdabra"` (note the capital `'A'`)? Would a simple `Expected 'a'` do? Or how about `Expected "cadabra"`? What if you wanted to choose a different error message along the lines of `"Magic word incorrect, try again!"`?

- Given `a or b`, if `a` fails on the input, do we *always* want to run `b`? Are there cases where we might not want to run `b`? If there are such cases, can you think of additional combinators that would allow the programmer to specify when `or` should consider the second parser?
- How do you want to handle reporting the *location* of errors?
- Given `a or b`, if `a` and `b` both fail on the input, should we support reporting both errors?

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

And do we *always* want to report both errors, or do we want to give the programmer a way to specify which of the two errors is reported?

SIDEBAR **Combinators specify information to implementation**

In a typical library design scenario where we have at least *some* idea of a concrete representation, we often think of functions in terms of how they will affect the final representation of our program.

By starting with the algebra first, we're forced to think differently—we must think of functions in terms of *what information they specify* to a possible implementation. The signatures determine what information is given to the implementation, and the implementation is free to use this information however it wants as long as it respects any specified laws.

9.5.1 First attempt at representing errors

Now that we have considered these ideas about error handling, we will start defining the algebra by progressively introducing our error-reporting combinators. Let's begin with an obvious one. None of the primitives so far let us assign an error message to a parser. We can introduce a primitive combinator for this called `tag`:

```
fun <A> tag(msg: String, p: Parser<A>): Parser<A>
```

The intended meaning of `tag` is that if `p` fails, its `ParseError` will somehow incorporate `msg`. What does this mean exactly? Well, we could do the simplest thing possible and assume that `ParseError` is simply a type alias for `String`, and that the returned `ParseError` will *equal* the tag. But we'd like our parse error to also tell us *where* the problem occurred. Let's tentatively add this concept to our algebra, call it `Location`.

Listing 9.7 Representing `ParseError` in terms of message and location

```
data class Location(val input: String, val offset: Int = 0) {

    private val slice by lazy { input.slice(0..offset + 1) }   ❶

    val line by lazy { slice.count { it == '\n' } + 1 }   ❷

    val column by lazy {
        when (val n = slice.lastIndexOf('\n')) {   ❸
            -1 -> offset + 1
            else -> offset - n
        }
    }
}

fun errorLocation(e: ParseError): Location

fun errorMessage(e: ParseError): String
```

❶ prepare a substring of the input up to where the error occurred

❷ calculate the number of lines to the error location

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❸ calculate the number of columns to the error location

We've picked a concrete representation for `Location` here that includes the full input, an offset into this input where the error occurred, and the line and column numbers which are computed lazily from the full input and offset. We can now say more precisely what we expect from `tag`. In the event of failure with `Left(e)`, `errorMessage(e)` will equal the message set by `tag`. What about the `Location`? We'd like for this to be provided by the `Parsers` implementation with the location of where the error occurred. This notion still seems a bit fuzzy at the moment—if we have `a or b`, and both parsers fail on the input, which location will be reported? In addition to that, which tag(s) will we see? We'll discuss this in greater depth in the following section on error nesting.

9.5.2 Accumulating errors through error nesting

Is the `tag` combinator sufficient for all our error-reporting needs? Not quite. Let's take a closer look with an example:

```
tag("first magic word", string("abra")) product   ❶
    string(" ").many() product   ❷
    tag("second magic word", string("cadabra"))   ❸
```

❶ Tag the first parser

❷ Skip any whitespace

❸ Tag the next parser

What sort of `ParseError` would we like to get back from `run(p, "abra cAdabra")`? Note the capital `A` in `cAdabra`. The immediate cause for error is this capital `'A'` instead of the expected lowercase `'a'`. That error has an *exact* location, and it will be helpful to report this somehow when debugging the issue. But reporting only that low-level error wouldn't be very informative, especially if this were part of a large grammar and we were running the parser on an even larger input.

When using `tag`, we should have some more contextual information—the immediate error occurred in the `Parser` tagged `"second magic word"`. This is certainly very helpful in pinpointing where things went wrong. Ideally, the error message should tell us that while parsing `"cAdabra"` using `"second magic word"`, there was an unexpected capital `'A'`. That highlights the error and gives us the context needed to understand it. Perhaps the top-level parser (`p` in this case) might be able to provide an even higher-level description of what the parser was doing when it failed, for instance `"parsing magic spell"`, which could also be informative.

So it seems wrong to assume that one level of error reporting will always be sufficient. Let's therefore provide a way to *nest* tags:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.8 The scope combinator is used for nesting tags

```
fun <A> scope(msg: String, p: Parser<A>): Parser<A>
```

Despite `scope` having the same method declaration as `tag`, the implementation of `scope` doesn't throw away the tag(s) attached to `p`—it merely adds *additional* information in the event that `p` fails. Let's specify what this means exactly. First, we modify the functions that pull information out of a `ParseError`. Rather than containing just a single `Location` and `String` message, we should rather get a `List<Pair<Location, String>>`:

Listing 9.9 Representing stacked errors using the ParseError data type

```
data class ParseError(val stack: List<Pair<Location, String>>)
```

This is a stack of error messages indicating what the `Parser` was doing when it failed. We can now specify what `scope` does when it encounters multiple errors—if `run(p, s)` is `Left(e1)`, then `run(scope(msg, p), s))` is `Left(e2)`, where `e2.stack.head` will contain `msg` and `e2.stack.tail` will contain `e1`.

We can write helper functions later to make constructing and manipulating `ParseError` values more convenient, and also to format them nicely for human consumption. For now we just want to make sure it contains all the relevant information for error reporting. For now, it does seems like `ParseError` will be sufficient for most purposes. Let's pick this as our concrete representation for use in the return type of `run` in the `Parsers` interface:

```
fun <A> run(p: Parser<A>, input: String): Either<ParseError, A>
```

9.5.3 Controlling branching and backtracking

There's one last concern regarding error reporting that we need to address. As we just discussed, when we have an error that occurs inside an `or` combinator, we need some way of determining which error(s) to report. We don't want to *only* have a global convention for this; we sometimes want to allow the programmer to control this choice. Let's look at a more concrete motivating example:

```
val spaces = string(" ").many()

val p1 = scope("magic spell") {
    string("abra") product spaces product string("cadabra")
}
val p2 = scope("gibberish") {
    string("abba") product spaces product string("babba")
}

val p = p1 or p2
```

What `ParseError` would we like to get back from `run(p, "abra cAdabra")`? Again, note the offending capital `A` in `cAdabra`. Both branches of the `or` will produce errors on the input. The

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`"gibberish"` parser will report an error due to expecting the first word to be `"abba"`, and the `"magic spell"` parser will report an error due to the accidental capitalization in `"cAdabra"`. Which of these errors do we want to report back to the user?

In this instance, we happen to want the `"magic spell"` parse error. After successfully parsing the `"abra"` word, we're *committed* to the `"magic spell"` branch of the `or`, which means if we encounter a parse error, we don't examine the subsequent branch of the `or`. In other instances, we may want to allow the parser to consider the next branch.

So it appears we need a primitive for letting the programmer indicate when to commit to a particular parsing branch. Recall that we loosely assigned `p1 or p2` to mean, *try running p1 on the input, and then try running p2 on the same input if p1 fails*. We can change its meaning to *try running p1 on the input, and if it fails in an uncommitted state, try running p2 on the same input; otherwise, report the failure*. This is useful for more than just providing good error messages—it also improves efficiency by letting the implementation avoid examining lots of possible parsing branches.

One common solution to this problem is to have all parsers *commit by default* if they examine at least one character to produce a result. We now introduce a combinator called `attempt`, which delays committing to a parse:

```
fun <A> attempt(p: Parser<A>): Parser<A>
```

It should satisfy something like the following situation. This is not *exactly* an equality; even though we want to run `p2` if the attempted parser `p1` fails, we may want `p2` to somehow incorporate the errors from *both* branches if it fails.

```
attempt(p1.flatMap { _ -> fail }) or p2 == p2
```

Here `fail` is a parser that *always* fails. In fact, we could introduce this as a primitive combinator if we like. What happens next is, even if `p1` fails midway through examining the input, `attempt` reverts the commit to that parse and allows `p2` to be run. The `attempt` combinator can be used whenever dealing with such ambiguous grammar. Multiple tokens may have to be examined before the ambiguity can be resolved and that parsing can commit to a single branch. As an example, we might write this:

```
(attempt(
    string("abra") product spaces product string("abra")
) product string("cadabra")) or
    (string("abra") product spaces product string("cadabra!"))
```

Suppose this parser is run on `"abra cadabra!"`. After parsing the first `"abra"`, we don't know whether to expect another `"abra"` (the first branch) or `"cadabra!"` (the second branch). By

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

wrapping an `attempt` around `string("abra") product spaces product string("abra")`, we allow the second branch to be considered up until we've finished parsing the second `"abra"`, at which point we commit to that branch.

EXERCISE 9.10

Can you think of any other primitives that might be useful for specifying what error(s) in an `or` chain get reported?

Note that we still haven't written an actual implementation of our algebra! Despite the lack of implementation, this process has been more about making sure that our combinators provide a well defined interface for the users of our library to interact with. More than that, it should provide a way for them to convey the right information to the underlying implementation. It will then be up to the implementation to interpret the information in a way that satisfies the laws we've stipulated.

9.6 Implementing the algebra

This entire chapter has been focused on building up an algebra of definitions without implementing a single thing! This has culminated in a final definition of `Parser<JSON>`. At this point it would be prudent to go back and retrofit the parser that you developed in exercise 9.9 with the error-reporting combinators discussed in section 9.5 if you haven't already done so. Now comes the exciting part where we define an implementation that can be run!

Our list of primitives has once again changed with the addition of our error handling combinators. Let's review the list one more time.

Table 9.3 Updated list of primitives to be used as basis for JSON parsing combinators

Primitive	Description
`string(s)`	Recognizes and returns a single `String`
`regex(s)`	Recognizes a regular expression of `String`
`slice(p)`	Returns the portion of input inspected by `p` if successful
`tag(msg, p)`	In the event of failure, replaces the assigned message with `msg`
`scope(msg, p)`	In the event of failure, adds `msg` to the error stack returned by `p`
`flatMap(p, f)`	Runs a parser, then uses its result to select a second parser to run in sequence
`attempt(p)`	Delays committing to `p` until after it succeeds
`or(p1, p2)`	Chooses between two parsers, first attempting `p1`, then `p2` if `p1` fails

The list has changed somewhat with the addition of `tag`, `scope` and `attempt`. We have also dropped `succeed` from our previous table 9.2.

In the next section, we'll work through a representation for `Parser` and implement the `Parsers` interface using this representation. The algebra we've designed places strong constraints on

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

possible representations. We should be able to come up with a simple, purely functional representation of `Parser` that can be used to implement the `Parsers` interface. But first, let's express the top level constructs that will be used as starting point for our implementation.

Listing 9.10 Top level representation of the parser combinator library

```
interface Parser<A>   ❶

data class ParseError(val stack: List<Pair<Location, String>>)

abstract class Parsers<PE> {   ❷
    internal abstract fun <A> or(p1: Parser<A>, p2: Parser<A>): Parser<A>
}

open class ParsersImpl<PE>() : Parsers<PE>() {   ❸
    override fun <A> or(p1: Parser<A>, p2: Parser<A>): Parser<A> = TODO()
}

abstract class ParserDsl<PE> : ParsersImpl<PE>() {   ❹
    infix fun <A> Parser<A>.or(p: Parser<A>): Parser<A> =
        this@ParserDsl.or(this, p)   ❷
}

object Example : ParserDsl<ParseError>() {   ❸
    init {
        val p1: Parser<String> = TODO()
        val p2: Parser<String> = TODO()
        val p3 = p1 or p2
    }
}
```

❶ Whatever representation we will discover for `Parser`, this is merely an example

❷ The `Parsers` class holds all the unimplemented primitive and combinator algebra definitions

❸ The `ParsersImpl` is the concrete implementation for all `Parsers`

❹ The `ParserDsl` adds syntactic sugar to make working with combinators easier

❺ Access the `or` function in `ParserImpl` through `this@ParserDsl`, workaround to prevent a circular reference.

❻ An object that makes use of our combinator library

9.6.1 Building up the algebra implementation gradually

We are finally going to discuss a concrete implementation of `Parsers` that fulfills all the accumulated features so far. Rather than jumping straight to the end with a final representation of `Parser`, we'll build it up gradually. We will do so by inspecting the primitives of the algebra, then reasoning about the information that will be required to support each one.

Let's begin with the `string` combinator:

```
fun string(s: String): Parser<String>
```

We also know we need to support the function `run`:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> run(p: Parser<A>, input: String): Either<PE, A>
```

As a first pass, we could assume that our `Parser` *is* simply the implementation of the `run` function:

```
typealias Parser<A> = (String) -> Either<ParseError, A>
```

We could use this to implement the `string` primitive as follows:

Listing 9.11 Implementation of the `string` primitive in terms of `Location`

```
override fun string(s: String): Parser<String> =
    { input: String ->
        if (input.startsWith(s))
            Right(s)
        else Left(Location(input).toError("Expected: $s"))   ❶
    }

private fun Location.toError(msg: String) =   ❷
    ParseError(listOf(Pair(this, msg)))
```

❶ Uses `toError` to construct a `ParseError`

❷ Extension that converts `Location` to a `ParseError`

The `else` branch of `string` has to build up a `ParseError`. These are a inconvenient to construct right now, so we've introduced a helper extension function called `toError` on `Location`.

9.6.2 Sequencing parsers after each other

So far, so good. We have a representation for `Parser` that at least supports `string`. Let's move on to sequencing of parsers. Unfortunately, to represent a parser like `"abra" product "cadabra"`, our existing representation isn't going to suffice. If the parse of `"abra"` is successful, then we want to consider those characters *consumed* before we run the `"cadabra"` parser on the remaining characters. So in order to support sequencing, we require a way of letting a `Parser` indicate how many characters it consumed. Capturing this turns out to be pretty easy, considering that `Location` contains the full input string *and* an offset into this string.

Listing 9.12 Representing the result as an ADT to track consumed characters

```
typealias Parser<A> = (Location) -> Result<A>   ❶

sealed class Result<out A>
data class Success<out A>(val a: A, val consumed: Int) : Result<A>()   ❷
data class Failure(val get: ParseError) : Result<Nothing>()
```

❶ The function definition of `Parser` now returns a `Result<A>`

❷ The `Success` type carries the count of consumed characters

We just introduced a richer alternative data type called `Result` instead of using a simple `Either` as before.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

In the event of success, we return a value of type `A`, as well as the number of characters of input consumed. The caller can then use this count to update the `Location` state. This type is starting to get to the essence of what a `Parser` truly is—it's a kind of state action that can fail, similar to what we built in chapter 6. It receives an input state, and on success returns a value as well as enough information to control how the state should be updated.

The understanding that a `Parser` is just a state action gives us a way of framing a representation that supports all the fancy combinators and laws we've stipulated so far. We simply consider what each primitive requires our state type to track, then we work through the details of how each combinator transforms this state.

EXERCISE **9.11 (Hard)**

Implement `string`, `regex`, `succeed`, and `slice` for this representation of `Parser`. Some private helper function stubs have been included to lead you in the right direction.

NOTE `slice` is probably less efficient than it could be, since it must still construct a value only to discard it. Don't bother addressing this as part of the current exercise.

EXERCISE

```
override fun string(s: String): Parser<String> = TODO()

private fun firstNonMatchingIndex(
    s1: String,
    s2: String,
    offset: Int
): Option<Int> = TODO()

private fun State.advanceBy(i: Int): State = TODO()

override fun regex(r: String): Parser<String> = TODO()

private fun String.findPrefixOf(r: Regex): Option<MatchResult> = TODO()

override fun <A> succeed(a: A): Parser<A> = TODO()

override fun <A> slice(p: Parser<A>): Parser<String> = TODO()

private fun State.slice(n: Int): String = TODO()
```

9.6.3 Capturing error messages through labeling parsers

Moving down our list of primitives, let's look at `scope` next. In the event of failure, we want to push a new message onto the `ParseError` stack. Let's introduce a helper function for this on `ParseError`. We'll call it `push` [48].

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.13 Extension function to push an error onto the `ParseError` stack head

```
fun ParseError.push(loc: Location, msg: String): ParseError =
    this.copy(stack = Pair(loc, msg) cons this.stack)
```

Now that we have this we can implement `scope` using the `mapError` extension method on `Result` that we will describe next:

Listing 9.14 Implementation of `scope` that records errors using `push`

```
fun <A> scope(msg: String, pa: Parser<A>): Parser<A> =
    { state -> pa(state).mapError { pe -> pe.push(state, msg) } }
```

The `mapError` extension method allows transformation of an error in case of failure, and is defined as follows:

Listing 9.15 Extension function to map `ParseError` on `Result` failure

```
fun <A> Result<A>.mapError(f: (ParseError) -> ParseError): Result<A> =
    when (this) {
        is Success -> this
        is Failure -> Failure(f(this.get))
    }
```

Because we push onto the stack after the inner parser has returned, the bottom of the stack will have more detailed messages that occurred later in parsing. For example, if `scope(msg1, a product scope(msg2, b))` fails while parsing `b`, the first error on the stack will be `msg1`, followed by whatever errors were generated by `a`, then `msg2`, and finally errors generated by `b`.

We can implement `tag` similarly, but instead of pushing onto the error stack, it replaces what's already there. We can write this again using `mapError` and an extension on `ParseError`, also called `tag`, which will be discussed afterwards:

Listing 9.16 Implementation of `tag` that records error using `tag`

```
fun <A> tag(msg: String, pa: Parser<A>): Parser<A> =
    { state ->
        pa(state).mapError { pe ->
            pe.tag(msg)   ❶
        }
    }
```

❶ Calls a helper method on `ParseError`, also named `tag`.

We added a helper extension function to `ParseError` that is also named `tag`. We'll make a design decision that `tag` trims the error stack, cutting off more detailed messages from inner scopes, using only the most recent location from the bottom of the stack. This is what it looks like:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.17 Extension function to tag `ParseError` on `Result` failure

```
fun ParseError.tag(msg: String): ParseError {

    val latest = this.stack.lastOrNone()  ❶

    val latestLocation = latest.map { it.first }  ❷

    return ParseError(latestLocation.map { Pair(it, msg) }.toList())  ❸
}
```

❶ Gets the last element of the stack or `None` if the stack is empty.

❷ Use its location if present

❸ Assemble a new `ParseError` with *only* this location and the tag message.

EXERCISE 9.12

Revise your implementation of `string` to provide a meaningful error message in the event of an error.

9.6.4 Recovering from error conditions and backtracking over them

Next, let's consider `or` and `attempt`. If we consider what we've already learned about `or`, we can summarize its behavior as follows: it should run the first parser, and if that fails *in an uncommitted state*, it should run the second parser on the same input. We also said that consuming at least one character should result in a *committed* parse, and that `attempt(p)` converts committed failures of `p` to *uncommitted* failures.

We can support the behavior we want by simply adding a field to the `Failure` case of `Result`. All we need is a `Boolean` value indicating whether the parser failed in a committed state, let's call it `isCommitted`:

```
data class Failure(
    val get: ParseError,
    val isCommitted: Boolean
) : Result<Nothing>()
```

The implementation of `attempt` now draws upon this new information and simply cancels the commitment of any failures that occur. It does so by using a helper function called `uncommit`, which we can define on `Result`:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.18 An implementation of `attempt` that cancels commitment of any failures

```
fun <A> attempt(p: Parser<A>): Parser<A> = { s -> p(s).uncommit() }

fun <A> Result<A>.uncommit(): Result<A> =
    when (this) {
        is Failure ->
            if (this.isCommitted)
                Failure(this.get, false)
            else this
        is Success -> this
    }
```

Now the implementation of `or` can simply check the `isCommitted` flag before running the second parser. Consider the parser `x or y`: if `x` succeeds, then the whole expression succeeds. If `x` fails in a *committed* state, we fail early and skip running `y`. Otherwise, if `x` fails in an *uncommitted* state, we run `y` and ignore the result of `x`:

Listing 9.19 An implementation of `or` that honors committed state

```
fun <A> or(pa: Parser<A>, pb: () -> Parser<A>): Parser<A> =
    { state ->
        when (val r: Result<A> = pa(state)) {
            is Failure ->
                if (!r.isCommitted) pb()(state)   ❶
                else r   ❷
            is Success -> r   ❸
        }
    }
```

❶ An *uncommitted* failure will invoke lazy `pb` and run it with original state passed to `or`

❷ A *committed* failure will pass through

❸ Success will pass through

9.6.5 Propagating state through context-sensitive parsers

Now for the final primitive in our list, `flatMap`. Recall that `flatMap` enables context-sensitive parsers by allowing the selection of a second parser to depend on the result of the first parser. The implementation is simple as we advance the location before calling the second parser. Again we will use a helper function, this time called `advanceBy` on `Location`. Despite this being simple, there is one caveat to be dealt with: if the first parser consumes any characters, we ensure that the second parser is committed using a helper function called `addCommit` on `ParseError`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 9.20 Ensuring that the parser is committed using `addCommit`

```
fun <A, B> flatMap(pa: Parser<A>, f: (A) -> Parser<B>): Parser<B> =
    { state ->
        when (val result = pa(state)) {
            is Success ->
                f(result.a)(state.advanceBy(result.consumed))   ❶
                    .addCommit(result.consumed != 0)   ❷
                    .advanceSuccess(result.consumed)   ❸
            is Failure -> result
        }
    }
```

❶ Advance the source location before calling second parser.

❷ Commit if the first parser has consumed any characters.

❸ Increment number of characters `consumed` to account for characters already consumed by `pa`.

In `advanceBy` on `Location`, we simply increment the offset:

```
fun Location.advanceBy(n: Int): Location =
    this.copy(offset = this.offset + n)
```

The `addCommit` function on `ParseError` is equally straightforward, ensuring that the committed state is updated if it was not already committed:

```
fun <A> Result<A>.addCommit(commit: Boolean): Result<A> =
    when (this) {
        is Failure ->
            Failure(this.get, this.isCommitted || commit)
        is Success -> this
    }
```

The final piece of the puzzle is the `advanceSuccess` function on `Result`, which is responsible for incrementing the number of consumed characters of a successful result. We want the total number of characters consumed by `flatMap` to be the sum of the consumed characters of the parser `pa` *and* the parser produced by `f`. We use `advanceSuccess` on the result of `f` to ensure that this adjustment is made:

```
fun <A> Result<A>.advanceSuccess(n: Int): Result<A> =
    when (this) {
        is Success ->
            Success(this.a, this.consumed + n)   ❶
        is Failure -> this   ❷
    }
```

❶ Advance the number of consumed characters by `n` on success

❷ Pass through the result on failure.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 9.13

Implement `run`, as well as any of the remaining primitives not yet implemented using our current representation of `Parser`. Try running your JSON parser on various inputs.

NOTE We should now have working code, although unfortunately you'll find that this causes a stack overflow for large inputs. Even though it is not required for this exercise, one simple solution to this problem is to provide a specialized implementation of `many` that avoids using a stack frame for each element of the list being built up. Ensuring that any combinators that do repetition are defined in terms of `many` (which they all can be), this solves the problem.

EXERCISE 9.14

Come up with a nice way of formatting a `ParseError` for human consumption. There are a lot of choices to make, but a key insight is that we typically want to combine or group tags attached to the same location when presenting the error as a `String` for display.

We could spend a lot more time improving and developing the example in this chapter, but we'll leave it at what it is for now. We do encourage you to keep playing and enhancing what you have on your own, though the parser combinator library itself really isn't the most important point that we're trying to bring home in this chapter—it was really all about demonstrating the approach of algebra-first library design.

This chapter concludes part 2 of the book. We hope that you've come away with an understanding of how to go about designing your own functional library. More importantly, we also hope that this section will inspire you to begin designing and building your *own* libraries based on domains that are of personal interest to you. Functional design isn't something reserved only for experts. It should be part of the day-to-day work done by functional programmers at all levels of experience.

Before you start on part 3, we implore you to venture out on your own by designing some libraries while writing functional code as you've been learning up to this point. Have lots of fun while you wrestle with, and conquer design problems that emerge as you go along. When you are done playing and come back, a universe of patterns and abstractions awaits you in part 3.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

9.7 Summary

- Algebraic library design establishes the interface with associated laws up front, then drives implementation as a result.
- A parser combinator library provides a motivating use case for functional library design and is well suited for an algebraic design approach.
- Primitives are simple combinators that don’t depend on others and provide building blocks for more complex higher order combinators.
- Algebraic design encourages invention of primitives first which allows discovery of more complex combinators to follow.
- A combinator is said to be context sensitive when it passes on state, so allowing the ability to sequence combinators.
- A parser combinator may accumulate errors, which allows for surfacing an error report in case of failure.
- A parser may fail with an uncommitted state which allows for backtracking and recovery from errors.
- Starting design with the algebra lets combinators specify information to implementation

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

10 Monoids

This chapter covers:

- An introduction to purely algebraic structures
- Learning about monoids and the laws they impose
- The connection between monoids and fold operations
- Using balanced folds to perform parallel computations in chunks
- Higher-kinded types and their application to foldable data structures
- Composing monoids to perform complex calculations from simpler parts

By the end of part 2, we were getting comfortable with considering data types in terms of their *algebras*. In particular, we were concerned with the operations they support and the laws that govern those operations. By now you will have noticed that the algebras of very different data types tend to share certain patterns in common. In this chapter, we'll begin identifying these patterns and taking advantage of them.

This chapter will be our first introduction to *purely algebraic structures*. As an example, we'll begin by considering a simple structure known as the *monoid*, which is defined *only by its algebra*. The name *monoid* might sound intimidating at first, but it is merely a mathematical term that in category theory refers to a *category with one object*. Other than satisfying the same laws, instances of the monoid interface may have little or nothing to do with one another. Nonetheless, we'll see how this algebraic structure is often all we need to write useful, polymorphic functions.

We choose to start with monoids because they're simple, ubiquitous, and useful. Monoids appear often in everyday programming whether we're aware of it or not. Working with lists, concatenating strings, or accumulating the results of a loop can often be phrased in terms of the monoid. In situations like this the use of *monoid instances* are employed as concrete implementations of this algebraic structure. We will begin with defining some monoid instances

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

for combining integers, booleans and `Option`s. We will then also see how monoid instances are a perfect fit for implementing `fold` operations on lists.

The chapter later culminates in how monoids can be used in two real-world situations: they facilitate parallelization by giving us the freedom to break problems into chunks that can be computed in parallel; they can also be composed to assemble complex calculations from simpler parts.

10.1 What is a monoid?

Grasping algebraic structures such as the monoid might seem like a daunting task, but when approaching it from a purely algebraic perspective we come to realize how simple it actually is. Rather than trying to explain it in words, we will first explore the concept by way of example.

Let's begin by considering the algebra of string concatenation. We can add `"foo" + "bar"` to get `"foobar"`. In addition to this, the empty string is known as an *identity element* for that operation. That is, if we say `(s + "")` or `("" + s)`, the result is always `s` for any value of `s`. Furthermore, if we combine three strings by saying `(r + s + t)`, the operation is *associative*. By this we mean that it doesn't matter whether we parenthesize it: `(r + s) + t` or `r + (s + t)`.

The exact same rules govern integer addition. It is associative since `(x + y) + z` is always equal to `x + (y + z)`. It has an identity element of `0` which does nothing when added to another integer. Ditto for multiplication, which works in the same way but has an identity element of `1`.

The Boolean operators `&&` and `||` are likewise associative, and they have identity elements `true` and `false`, respectively.

These are just a few simple examples, but when you go looking, algebras like this can be found everywhere where laws apply. The term for this particular kind of algebra is *monoid*, and the laws of associativity and identity are collectively called the *monoid laws*. A monoid consists of the following:

- Some type `A`
- An associative binary operation `combine`, that takes two values of type `A` and combines them into one: `combine(combine(x,y), z) == combine(x, combine(y,z))` for any choice of `x: A`, `y: A`, `z: A`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 10.1 The law of associativity expressed in terms of combine for monoids

- The value `nil: A`, that is an identity for that operation: `combine(x, nil) == x` and `combine(nil, x) == x` for any `x: A`

Figure 10.2 The law of identity expressed in terms of combine for monoids

We can express this in terms of a Kotlin interface:

Listing 10.1 A monoid expressed as a Kotlin interface

```
interface Monoid<A> {
    fun combine(a1: A, a2: A): A   ❶
    val nil: A   ❷
}
```

❶ Satisfies the law of *associativity*, `combine(combine(x, y), z) == combine(x, combine(y, z))`

❷ Satisfies the law of *identity*, `combine(x, nil == x` and `combine(nil, x) == x`

An example instance of this interface is the `String` monoid:

```
val stringMonoid = object : Monoid<String> {

    override fun combine(a1: String, a2: String): String = a1 + a2
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    override val nil: String = ""
}
```

List concatentation also forms a monoid, this method is able to generate a monoid for any type `A`:

```
fun <A> listMonoid(): Monoid<List<A>> = object : Monoid<List<A>> {

    override fun combine(a1: List<A>, a2: List<A>): List<A> = a1 + a2

    override val nil: List<A> = emptyList()
}
```

SIDEBAR **The purely abstract nature of an algebraic structure**

Notice that other than satisfying the monoid laws, the various `Monoid` instances don't have much to do with each other. The answer to the question "What is a monoid?" is simply that a monoid is a type, together with the monoid operations and a set of laws. A monoid is the algebra, and nothing more. Of course, you may build some other intuition by considering the various concrete instances, but this intuition isn't necessarily correct and nothing guarantees that all monoids you encounter will match your intuition!

EXERCISE **10.1**

Give `Monoid` instances for integer addition and multiplication, as well as for Boolean operators.

```
val intAddition: Monoid<Int> = TODO()

val intMultiplication: Monoid<Int> = TODO()

val booleanOr: Monoid<Boolean> = TODO()

val booleanAnd: Monoid<Boolean> = TODO()
```

EXERCISE **10.2**

Give a `Monoid` instance for combining `Option` values.

```
fun <A> optionMonoid(): Monoid<Option<A>> = TODO()

fun <A> dual(m: Monoid<A>): Monoid<A> = TODO()
```

EXERCISE **10.3**

A function having the same argument and return type is sometimes called an *endofunction*.[49] Write a monoid for endofunctions.

```
fun <A> endoMonoid(): Monoid<(A) -> A> = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 10.4

Use the property-based testing framework we developed in chapter 8 to implement properties for the monoid laws of associativity and identity. Use your properties to test some of the monoids we've written so far.

```
fun <A> monoidLaws(m: Monoid<A>, gen: Gen<A>): Prop = TODO()
```

SIDEBAR Talking about monoids

Programmers and mathematicians disagree about terminology whey they talk about a type *being* a monoid versus *having* a monoid instance.

As a programmer, it is tempting to think of the instance of type `Monoid<A>` as *being* a monoid. But that isn't really true. The monoid is actually both things—the type together with the instance satisfying the laws. It's more accurate to say that the type `A` *forms* a monoid under the operations defined by the `Monoid<A>` instance. Put in a different way, we might say that "type `A` *is* a monoid," or even that "type `A` is *monoidal*." In any case, the `Monoid<A>` instance is simply evidence of this fact.

This is much the same as saying that the page or screen you're reading "forms a rectangle" or "is rectangular." It's less accurate to say that it "is a rectangle" (although that still makes sense), but to say that it "has a rectangle" would be strange.

Just what *is* a monoid, then? It's simply a type `A` and an implementation of `Monoid<A>` that satisfies the laws. Stated otherwise, *a monoid is a type together with a binary operation (`combine`) over that type, satisfying associativity and having an identity element (`nil`)*.

What does this buy us? Just like any abstraction, a monoid is useful to the extent that we can write useful generic code assuming only the capabilities provided by the abstraction. Can we write any interesting programs, knowing nothing about a type other than that it forms a monoid? Absolutely! Let's look at some examples.

10.2 Folding lists with monoids

Monoids have an intimate connection with lists. If we recall the various fold operations that were defined on the `List` type in chapter 3, we see that two parameters were always present: a zero value initializer, and a function that combined two values into an accumulated result. All of this was done in context of a single type, namely that of the initializer value.

Let's take a closer look at the signatures of of `foldLeft` and `foldRight` on `List` to confirm the above observation.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A, B> foldRight(z: B, f: (A, B) -> B): B

fun <A, B> foldLeft(z: B, f: (B, A) -> B): B
```

We see the initializer `z`, the combining function `(A, B) -> B` and the result type of `B` carried through from the initializer.

What happens if we turn `A` and `B` into a single type by calling it `A`?

```
fun <A> foldRight(z: A, f: (A, A) -> A): A

fun <A> foldLeft(z: A, f: (A, A) -> A): A
```

The components of a monoid fit these argument types like a glove. So if we had a `List<String>`, `words`, we could simply pass the `combine` and `nil` of the `stringMonoid` in order to reduce the list with the monoid and concatenate all the strings. Let's try this in the REPL:

```
>>> val words = listOf("Hic", "Est", "Index")
res0: kotlin.collections.List<kotlin.String> = [Hic, Est, Index]

>>> words.foldRight(stringMonoid.nil, stringMonoid::combine)
res1: kotlin.String = HicEstIndex

>>> words.foldLeft(stringMonoid.nil, stringMonoid::combine)
res2: kotlin.String = HicEstIndex
```

Note that it doesn't matter if we choose `foldLeft` or `foldRight` when folding with a monoid. We should get the same result in both cases. This is precisely because the laws of associativity and identity hold. A left fold associates operations to the left, whereas a right fold associates to the right, with the identity element on the far left and right respectively:

```
>>> words.foldLeft("") { a, b -> a + b } == (("" + "Hic") + "Est") + "Index"
res3: kotlin.Boolean = true

>>> words.foldRight("") { a, b -> a + b } == "Hic" + ("Est" + ("Index" + ""))
res4: kotlin.Boolean = true
```

Armed with this knowledge, we can now write a function called `concatenate` that folds a list with a monoid.

```
fun <A> concatenate(la: List<A>, m: Monoid<A>): A =
    la.foldLeft(m.nil, m::combine)
```

In some circumstances the element type might not have a `Monoid` instance. In cases like this we could always `map` over the list to turn it into a type that does have an associated instance.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 10.5

The function `foldMap` is used to align the types of the list elements so that a `Monoid` instance may be applied to the list. Implement this function.

```
fun <A, B> foldMap(la: List<A>, m: Monoid<B>, f: (A) -> B): B = TODO()
```

EXERCISE 10.6 (Hard)

The `foldMap` function can be implemented using either `foldLeft` or `foldRight`. But you can also write `foldLeft` and `foldRight` using `foldMap`. Give it a try for fun!

```
fun <A, B> foldRight(la: Sequence<A>, z: B, f: (A, B) -> B): B = TODO()

fun <A, B> foldLeft(la: Sequence<A>, z: B, f: (B, A) -> B): B = TODO()
```

10.3 Associativity and parallelism

Processing a list sequentially from the left or right is not very efficient when we want to *parallelize* such a process. This becomes increasingly important as the list size grows and the computation becomes more complex. It is possible to take advantage of the associative aspect of the monoid to come up with a more efficient way of folding such lists. In this section, we will explore how to do this using a technique called the *balanced fold*. This utilizes the monoid to achieve a more efficient fold that can in turn be used in parallel computation.

But what exactly is a balanced fold? Let's look at this by way of example.

Suppose we have a sequence `a`, `b`, `c`, `d` that we'd like to reduce using some monoid. Folding to the right, the combination of `a`, `b`, `c`, and `d` would look like this:

```
combine(a, combine(b, combine(c, d)))
```

Folding to the left would look like this:

```
combine(combine(combine(a, b), c), d)
```

But a balanced fold looks like this:

```
combine(combine(a, b), combine(c, d))
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 10.3 A balanced fold splits the workload into equal groups for efficient processing

Note that the balanced fold allows for parallelism because the two inner `combine` calls are independent and can be run simultaneously. But beyond that, the more balanced tree structure can be more efficient in cases where the cost of each `combine` is proportional to the size of its arguments. For instance, consider the runtime performance of this expression:

```
listOf("lorem", "ipsum", "dolor", "sit")
    .foldLeft("") { a, b -> a + b }
```

At every step of the fold, we're allocating the full intermediate `String` only to discard it and allocate a larger string in the next step. Recall that `String` values are immutable, and that evaluating `a + b` for strings `a` and `b` requires allocating a fresh character array and copying both `a` and `b` into this new array. It takes time proportional to `a.length + b.length`.

We can confirm this by tracing through the evaluation of the preceding expression:

```
listOf("lorem", "ipsum", "dolor", "sit")
    .foldLeft("") { a, b -> a + b }

listOf("ipsum", "dolor", "sit")
    .foldLeft("lorem") { a, b -> a + b }

listOf("dolor", "sit")
    .foldLeft("loremipsum") { a, b -> a + b }

listOf("sit")
    .foldLeft("loremipsumdolor") { a, b -> a + b }

listOf<String>()
    .foldLeft("loremipsumdolorsit") { a, b -> a + b }

"loremipsumdolorsit"
```

Note the intermediate strings being created in each step and then immediately being discarded. A more efficient strategy would be the balanced fold as described earlier. Here we combine the sequence in halves, first constructing `"loremipsum"` and `"dolorsit"`, then finally adding those together to form `"loremipsumdolorsit"`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 10.7

Implement `foldMap` based on the *balanced fold* technique. Your implementation should use the strategy of splitting the sequence in two, recursively processing each half, and then adding the answers together using the provided monoid.

```
fun <A, B> foldMap(la: List<A>, m: Monoid<B>, f: (A) -> B): B = TODO()
```

EXERCISE 10.8

Also implement a *parallel* version of `foldMap` called `parFoldMap` using the library we developed in chapter 7.

TIP

Implement `par`, a combinator to promote `Monoid<A>` to a `Monoid<Par<A>>`, and then use this to implement `parFoldMap`.

EXERCISE

```
fun <A> par(m: Monoid<A>): Monoid<Par<A>> = TODO()

fun <A, B> parFoldMap(
    la: List<A>,
    pm: Monoid<Par<B>>,
    f: (A) -> B
): Par<B> = TODO()
```

EXERCISE 10.9 (Hard/Optional)

Use `foldMap` as developed in exercise 10.7 to detect ascending order of a `List<Int>`. This will require some creativity when deriving the appropriate `Monoid` instance.

```
fun ordered(ints: Sequence<Int>): Boolean = TODO()
```

10.4 Example: Parallel parsing

Up to this point, we've been looking at trivial examples that have little or no use in your day-to-day work. Even though asserting list ordering might be mildly useful, we are going to apply this to a non-trivial use case like you would encounter in the real world. A good example of such a case is a word count program.

For our example, let's say that we wanted to count the number of words in a `String`. This is a fairly simple parsing problem. We could scan the string character by character, looking for whitespace and counting up the number of runs of consecutive non-whitespace characters.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Parsing sequentially like that, the parser state could be as simple as tracking whether the last character seen was a whitespace.

This is well and good for a short string, but imagine doing this for an enormous text file possibly too big to fit in memory on a single machine. It would be great if we could work with chunks of the file in parallel. The strategy would be to split the file into manageable chunks, process several chunks in parallel, and then combine the results. In that case, the parser state needs to be slightly more complicated, and we need to be able to combine intermediate results regardless of whether the section we're looking at is at the beginning, middle, or end of the file. In other words, we want the combining operation to be associative.

To keep things simple and concrete, let's consider a short string and pretend it's a large file:

```
"lorem ipsum dolor sit amet, "
```

If we split this string roughly in half, we might split it in the middle of a word. In the case of our string, that would yield `"lorem ipsum do"` and `"lor sit amet, "`. When we add up the results of counting the words in these strings, we want to avoid double-counting the word `dolor`. Clearly, just counting the words as an `Int` isn't sufficient. We need to find a data structure that can handle partial results like the half words `do` and `lor`, and can track the complete words seen so far, like `ipsum`, `sit`, and `amet`. We can represent this using the following algebraic data type.

Listing 10.2 ADT representation of partial results of a word count

```
sealed class WC

data class Stub(val chars: String) : WC()   ❶
data class Part(val ls: String, val words: Int, val rs: String) : WC()  ❷
```

❶ A `Stub` is an accumulation of characters that form a partial word

❷ A `Part` contains a left stub, word count and right stub.

A `Stub` is the simplest case where we haven't seen any complete words yet. A `Part` keeps the count of complete words we've seen so far as integer `words`. The value `ls` holds any partial word we've seen to the left of those words, and `rs` holds any partial word to the right.

For example, counting over the string `"lorem ipsum do"` would result in `Part ("lorem", 1, "do")` since there's one certainly complete word, `"ipsum"`. And since there's no whitespace to the left of `lorem` or right of `do`, we can't be sure if they're complete words or not so we don't count them. Counting over `"lor sit amet, "` would result in `Part("lor", 2, "")`, discarding the comma.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 10.10

Write a monoid instance for `WC` and ensure that it meets both monoid laws.

```
val wcMonoid: Monoid<WC> = TODO()
```

EXERCISE 10.11

Use the `WC` monoid to implement a function that counts words in a `String` by recursively splitting it into substrings and counting the words in those substrings.

```
fun wordCount(s: String): Int = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR **Monoid homomorphisms**

If you've donned your law-discovering hat while reading this chapter, you may have noticed that a law exists holding for some functions *between* monoids.

For instance, take the `String` concatenation monoid and the integer addition monoid. If you take the lengths of two strings and add them up, it's the same as taking the length of the concatenation of those two strings:

```
"foo".length + "bar".length == ("foo" + "bar").length
```

Here, `length` is a function from `String` to `Int` that *preserves the monoid structure*. Such a function is called a *monoid homomorphism*.[50] A monoid homomorphism `f` between monoids `M` and `N` obeys the following general law for all values `x` and `y`:

```
M.combine(f(x), f(y)) == f(N.combine(x, y))
```

The same law should hold for the homomorphism from `String` to `WC` in the preceding exercises.

This property can be useful when designing your own libraries. If two types that your library uses are monoids, and some functions exist between them, it's a good idea to think about whether those functions are expected to preserve the monoid structure and to check the monoid homomorphism law with property-based tests.

Sometimes there will be a homomorphism in both directions between two monoids. If they satisfy a *monoid isomorphism* (*iso-* meaning *equal*), we say that the two monoids are isomorphic. A monoid isomorphism between `M` and `N` has two homomorphisms `f` and `g`, where both `f andThen g` and `g andThen f` are an identity function.

For example, the `String` and `Array<Char>` monoids are isomorphic in terms of concatenation . The two Boolean monoids `(false, ||)` and `(true, &&)` are also isomorphic, via the `!` (negation) operation.

10.5 Foldable data structures

In chapter 3 we implemented the data structures `List` and `Tree`, both of which could be folded. Then in chapter 5 we wrote `Stream`, a lazy structure that could also be folded much like `List`. As if that wasn't enough, we've now added fold functionality that operates on Kotlin's `Sequence` too.

When we're writing code that needs to process data contained in one of these structures, we often don't care about the *shape* of the structure. It doesn't matter if it's a tree, list, lazy, eager, efficiently random access, and so forth.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

For example, if we have a structure full of integers and want to calculate their sum, we can use `foldRight`:

```
ints.foldRight(0) { a, b -> a + b }
```

Looking at this code snippet, we shouldn't have to care about the type of `ints` at all. It could be a `Vector`, a `Stream`, a `List`, or anything at all with a `foldRight` method for that matter. We can capture this commonality in the following interface for all these container types:

```
interface Foldable<F> {  ❶

    fun <A, B> foldRight(fa: Kind<F, A>, z: B, f: (A, B) -> B): B  ❷

    fun <A, B> foldLeft(fa: Kind<F, A>, z: B, f: (B, A) -> B): B

    fun <A, B> foldMap(fa: Kind<F, A>, m: Monoid<B>, f: (A) -> B): B

    fun <A> concatenate(fa: Kind<F, A>, m: Monoid<A>): A =
        foldLeft(fa, m.nil, m::combine)
}
```

❶ The interface declares type `F` that represents any container

❷ The `Kind<F, A>` represents the kind of `F<A>`

The `Foldable` interface declares a generic type of `F`, which could represent any container such as `Option`, `List` or `Stream`. We also see something new by way of a type called `Kind<F, A>`, which represents the `F<A>`. We can't express multiple levels of generics in Kotlin type declarations, so Arrow provides us with `Kind` to declare that the kind of `F` is an outer container for inner elements of type `A`. Just like functions that take other functions as arguments are called higher-order functions, something like `Foldable` is a *higher-order type constructor* or a *higher-kinded type*.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR

Higher-kinded types and Kotlin

If you've come from an object oriented programming background, you will know what a constructor is. In particular, a *value constructor* is a method or function that has a value applied to it to "construct" another value (object). Likewise, we have something called a *type constructor*, which is a type that allows another type to be applied to it. The result of this construction is called a *higher-kinded type*.

As an example, take the `Foldable` interface. We declare a new instance of this interface which is a `ListFoldable`, a `Foldable` of the `List` type. Let's express this exact situation with a snippet of pseudocode:

```
interface Foldable<F<A>> {
    //some abstract methods
}

object ListFoldable : Foldable<List<A>> {
    //some method implementations with parametrized A
}
```

On closer inspection, this is not as simple as what we would have thought. Here we are dealing with a type constructor that is a `Foldable` of `F<A>`, which in the implementation is a `List<A>`, but could also be a `Stream<A>`, `Option<A>` or something else. Notice the two levels of generics that we are dealing with, namely `F` and `A`, or more concretely, `List<A>` in the implementation. *This nesting of kinds can't be expressed in Kotlin and will fail compilation.*

Higher-kinded types are an advanced language feature which languages like Kotlin and Java do not support. Although this might change in the future, the Arrow team have provided an interim workaround for situations like this. Appendix D goes into greater detail about how Arrow solves this problem to enable higher-kinded types in Kotlin.

NOTE

To reiterate, Kotlin is not able to express higher-kinded types directly, and so we need to rely on Arrow to give us this ability. **Please ensure that you have read and have understood appendix D before continuing. All subsequent material builds upon this knowledge.**

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 10.12

Implement `foldLeft`, `foldRight` and `foldMap` on the `Foldable<F>` interface in terms of each other. It is worth mentioning that using these functions in terms of each other could result in undesired effects like circular references. This will be addressed in exercise 10.13.

```
interface Foldable<F> {

    fun <A, B> foldRight(fa: Kind<F, A>, z: B, f: (A, B) -> B): B = TODO()

    fun <A, B> foldLeft(fa: Kind<F, A>, z: B, f: (B, A) -> B): B = TODO()

    fun <A, B> foldMap(
        fa: Kind<F, A>,
        m: Monoid<B>,
        f: (A) -> B
    ): B = TODO()
}
```

EXERCISE 10.13

Implement `Foldable<ForList>` using the `Foldable<F>` interface from the previous exercise.

```
object ListFoldable : Foldable<ForList>
```

EXERCISE 10.14

Recall that we implemented a binary `Tree` in chapter 3. Next, implement `Foldable<ForTree>`. You only need to override `foldMap` of `Foldable` to make this work, letting the provided `foldLeft` and `foldRight` methods use your new implementation.

NOTE A foldable version of `Tree`, along with `ForTree` and `TreeOf` has been provided in the chapter 10 exercise boilerplate code.

EXERCISE

```
object TreeFoldable : Foldable<ForTree>
```

SIDEBAR The semigroup and its relation to monoid

We have started part 3 of the book with monoids because they are simple and easy to understand. Despite their simplicity, they can be broken down even further into smaller units called *semigroups*.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Figure 10.4 The semigroup encompasses combinatorial aspect of the monoid

SIDEBAR

As we have already learned, the monoid consists of two operations, namely an ability to *combine*, and another to create an empty *nil* value. The ability to combine is known as a semigroup and can be defined as follows:

```
interface Semigroup<A> {
    fun combine(a1: A, a2: A): A
}
```

In other words, a monoid is the combination of a semigroup with a nil value operation, and can be expressed as follows:

```
interface Monoid<A> : Semigroup<A> {
    val nil: A
}
```

Even though we won't be using the semigroup directly, it is still good to know that the monoid is not the simplest algebraic structure available.

EXERCISE 10.15

Write an instance of `Foldable<ForOption>`.

```
object OptionFoldable : Foldable<ForOption>
```

EXERCISE 10.16

Any `Foldable` structure can be turned into a `List`. Write this convenience method for `Foldable<F>` using an existing method on the interface.

```
fun <A> toList(fa: Kind<F, A>): List<A> = TODO()
```

10.6 Composing monoids

The monoids we have covered up to now were self-contained and didn't depend on other monoids for their functionality. This section will deal with monoids that depend on other monoids to implement their functionality.

When considering the monoid by itself, its applications are rather limited. Next, we will look at

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

ways that we can make it more useful by combining it with other monoids. We can achieve this by either *composing* or *nesting* monoids.

The `Monoid` abstraction in itself is not all that compelling, and with the generalized `foldMap` it's only vaguely more interesting. The real power of monoids comes from the fact that they *compose* . In other words, if types `A` and `B` are both monoids, then they can be composed as a new monoid of `Pair<A, B>`. We refer to this monoidal composition as their *product*.

EXERCISE 10.17

Implement the `productMonoid` by composing two monoids. Your implementation of `combine` should be associative as long as `A.combine` and `B.combine` are both associative.

```
fun <A, B> productMonoid(
    ma: Monoid<A>,
    mb: Monoid<B>
): Monoid<Pair<A, B>> = TODO()
```

10.6.1 Assembling more complex monoids

One way to enhance monoids is to let them depend on one another. This section will deal with assembling monoids from other monoids.

Some data structures form interesting monoids as long as the types of the elements they contain also form monoids. For instance, there's a monoid for merging key-value pairs of `Maps`, as long as the value type is a monoid.

Listing 10.3 A monoid that merges key-value maps through the use of another monoid

```
fun <K, V> mapMergeMonoid(v: Monoid<V>): Monoid<Map<K, V>> =
    object : Monoid<Map<K, V>> {
        override fun combine(a1: Map<K, V>, a2: Map<K, V>): Map<K, V> =
            (a1.keys + a2.keys).foldLeft(nil, { acc, k ->
                acc + mapOf(
                    k to v.combine(
                        a1.getOrDefault(k, v.nil),
                        a2.getOrDefault(k, v.nil)
                    )
                )
            })

        override val nil: Map<K, V> = emptyMap()
    }
```

We can now assemble complex monoids easily by using this simple `mapMergeMonoid` combinator as follows:

```
val m: Monoid<Map<String, Map<String, Int>>> =
    mapMergeMonoid<String, Map<String, Int>>(
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        mapMergeMonoid<String, Int>(
            intAdditionMonoid
        )
    )
```

This allows us to combine nested expressions using the monoid with no additional programming. Let's take this to the REPL:

```
>>> val m1 = mapOf("o1" to mapOf("i1" to 1, "i2" to 2))
>>> val m2 = mapOf("o1" to mapOf("i3" to 3))

>>> m.combine(m1, m2)

res0: kotlin.collections.Map<kotlin.String,kotlin.collections.Map<
    kotlin.String, kotlin.Int>> = {o1={i1=1, i2=2, i3=3}}
```

By nesting monoids, we have now been able to merge a nested data structure by issuing a single command. Next, we will look at a monoid that emits a function as a monoid.

EXERCISE 10.18

Write a monoid instance for functions whose results themselves are monoids.

```
fun <A, B> functionMonoid(b: Monoid<B>): Monoid<(A) -> B> = TODO()
```

EXERCISE 10.19

A bag is like a set, except that it's represented by a map that contains one entry per element with that element as the key, and the value under that key is the number of times the element appears in the bag. For example:

```
>>> bag(listOf("a", "rose", "is", "a", "rose"))

res0: kotlin.collections.Map<kotlin.String, kotlin.Int> = {a=2, rose=2, is=1}
```

Use monoids to compute such a bag from a `List<A>`.

```
fun <A> bag(la: List<A>): Map<A, Int> = TODO()
```

10.6.2 Using composed monoids to fuse traversals

Sometimes we require several calculations to be applied to a list, which would normally result in multiple traversals to get the results. This section will describe how we can use monoids to perform these calculations simultaneously during a single traversal.

The fact that multiple monoids can be composed into one means that we can perform multiple calculations together when folding a data structure. For example, we can take the length and sum of a list simultaneously in order to calculate the mean:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 10.4 Determine the mean of a list by calculating the length and sum simultaneously

```
>>> val m = productMonoid<Int, Int>(intAdditionMonoid, intAdditionMonoid)
>>> val p = ListFoldable.foldMap(List.of(1, 2, 3, 4), m, { a -> Pair(1, a) })

res0: kotlin.Pair<kotlin.Int, kotlin.Int> = (4, 10)

>>> val mean = p.first / p.second.toDouble()
>>> mean

res1: kotlin.Double = 0.4
```

It can be tedious to assemble monoids by hand using `productMonoid` and `foldMap`. Part of the problem is that we're building up the `Monoid` separately from the mapping function of `foldMap`, and we must manually keep these "aligned" as we did here. A better way would be to create a combinator library that makes it more convenient to assemble these composed monoids and define complex computations that may be parallelized and run in a single pass. Such a library is beyond the scope of this chapter, but worth exploring as a fun project if this fascinates you.

Our goal in part 3 is to get you accustomed to working with more abstract structures, and to develop the ability to recognize them. In this chapter, we introduced one of the simplest purely algebraic abstractions, the monoid. When you start looking for it, you'll find ample opportunity to exploit the monoidal structure in your own libraries. The associative property enables folding any `Foldable` data type and gives the flexibility of doing so in parallel. Monoids are also compositional, and you can use them to assemble folds in a declarative and reusable way.

`Monoid` has been our first purely abstract algebra, defined only in terms of its abstract operations and the laws that govern them. We saw how we can still write useful functions that know nothing about their arguments except that their type forms a monoid. This more abstract mode of thinking is something we'll develop further in the rest of part 3. We'll consider other purely algebraic interfaces and show how they encapsulate common patterns that we've repeated throughout this book.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

10.7 Summary

- A *purely algebraic structure* is a declarative abstraction of laws that enforces these laws when writing polymorphic functions.
- The *monoid* is an algebraic structure that upholds the laws of *associativity and identity*, and is defined by a type with operations that uphold these laws.
- Monoid operations are closely related to fold operations, and are most often used for such operations.
- *Balanced folds* are highly effective in parallelization and are a natural fit for monoids.
- *Higher-kinded types* allow abstraction of operations to promote code reuse across multiple implementations, and are types which take other types to construct new types.
- Monoids may be composed to form *products* that represent more complex monoidal structures.
- Multiple operations can be applied simultaneously with composed monoids, so preventing unnecessary list traversal.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

11 Monads and functors

This chapter covers:

- Defining the functor by generalizing the map operation
- Deriving general purpose methods by applying the functor
- Revisiting and formalizing the functor law
- Defining various combinators that constitute the monad
- Introducing and proving the laws that govern the monad
- Discovering the true meaning of the monad

Many developers break out in a cold sweat on hearing the term *monad*. We have visions of people sitting in their lofty ivory towers, completely disconnected from the reality that we live in. We hear them mumbling away about academic concepts that have little or no bearing on the real world.

Even though individuals have used the term *monad* in such ways in the past, we hope to show that this can be no further from the truth. The monad concept is highly practical in its application, and can truly transform the way we write code. Granted, this term (along with its relative the *functor*, which we will also come to know in this chapter) does find its origins in the academic roots of category theory. Despite this, we will learn that it's highly practical and nothing to fear.

This chapter serves to demystify the ominous monad, and by the end of it you should have a working understanding of what it is, as well as how to apply it in a pragmatic way to your daily programming challenges. This could well be one of the most important lessons to learn from this book.

Chapter 10 introduced a simple algebraic structure, the monoid. This was our first instance of a completely abstract, purely algebraic interface, and it led us to think about interfaces in a new

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

way to how we have viewed them in an object oriented way. That is, a useful interface may be defined only by a collection of operations related by laws.

This chapter will continue this mode of thinking and apply it to the problem of factoring out code duplication across some of the libraries we wrote in parts 1 and 2. We'll discover two new abstract interfaces, `Monad` and `Functor`, and get more general experience with spotting these sorts of abstract structures in our code.

11.1 Functors: generalizing the `map` function

The focal point of this chapter is the monad, but to fully grasp what it's about we need to come to terms with the functor on which it relies. In chapter 10 we learned that monoid had a relationship with the semigroup. In fact, we discovered that the monoid *is* a semigroup with some additional functionality[51]. Although the relationship between the monad and functor isn't as clear cut as this, we can still say that a monad is usually a functor too. For this reason this section will help us first understand what a functor is and how to apply it. Once we have laid this foundation, we can advance into the territory of monads with confidence.

In parts 1 and 2 we implemented several different combinator libraries. In each case, we proceeded by writing a small set of primitives and then a number of combinators defined purely in terms of those primitives. We noted some similarities between derived combinators across the libraries we wrote. For instance, we implemented a `map` function for each data type, to lift a function transforming one argument "in context of" some data type. For `Option`, `Gen`, and `Parser`, the type signatures were as follows:

```
fun <A, B> map(ga: Option<A>, f: (A) -> B): Option<B>

fun <A, B> map(ga: Gen<A>, f: (A) -> B): Gen<B>

fun <A, B> map(ga: Parser<A>, f: (A) -> B): Parser<B>
```

These type signatures differ only in the concrete data type (`Option`, `Gen` or `Parser`). We can capture this idea with a Kotlin interface called `Functor` as *a data type that implements `map`*:

Listing 11.1 The functor interface for kind of `F` defines `map` functionality

```
interface Functor<F> {
    fun <A, B> map(fa: Kind<F, A>, f: (A) -> B): Kind<F, B>
}
```

Here we've parameterized `map` on the type constructor, `Kind<F, A>`, much like we did with `Foldable` in chapter 10. Recall that a *type constructor* is applied to a type to produce another type. For example, `List` is a type constructor, not a type. There are no values of type `List`, but we can apply it to the type `Int` to produce the type `List<Int>`. Likewise, `Parser` can be applied

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

to `String` to yield `Parser<String>`. Instead of picking a particular `Kind<F, A>`, like `Gen<A>` or `Parser<A>`, the `Functor` interface is parametric in the choice of `F`. Here's an instance for `List`:

```
val listFunctor = object : Functor<ForList> {
    override fun <A, B> map(fa: ListOf<A>, f: (A) -> B): ListOf<B> =
        fa.fix().map(f)
}
```

NOTE As in chapter 10, we draw on the `Kind` type along with it's related boilerplate code to express higher kinded types in Kotlin. Please refer to Appendix D to understand what this entails.

We say that a type constructor like `List` (or `Option`, or `F`) is a functor, and the `Functor<F>` instance constitutes proof that `F` is in fact a functor.

What can we do with this abstraction? We can discover useful functions just by having a *play* with the operations of the interface in a purely algebraic way. Let's see what (if any) useful operations we can define only in terms of `map`. Let's look at such an example. If we have `F<Pair<A, B>>` where `F` is a functor, we can "distribute" the `F` over the pair to get `Pair<F<A>, F<B>>`:

```
fun <A, B> distribute(
    fab: Kind<F, Pair<A, B>>
): Pair<Kind<F, A>, Kind<F, B>> =
    Pair(map(fab) { it.first }, map(fab) { it.second })
```

We wrote this by merely following the types, but let's think about what it *means* for concrete data types like `List`, `Gen`, `Option`, and so on. For example, if we `distribute` a `List<Pair<A, B>>`, we get two lists of the same length, one with all the `A`s and the other with all the `B`s. That operation is sometimes called *unzip*. So we just wrote a generic unzip function that works not just for lists, but for any functor!

And when we have an operation on a product like this, we should see if we can construct the opposite operation over a sum or *coproduct*. A coproduct is the term in category theory given to a *disjoint union*, or `Either`, as we have come to know it so far. In our case, given a coproduct of higher kinds, we should get back a kind of coproducts. We will call this `codistribute`.

```
fun <A, B> codistribute(
    e: Either<Kind<F, A>, Kind<F, B>>
): Kind<F, Either<A, B>> =
    when (e) {
        is Left -> map(e.a) { Left(it) }
        is Right -> map(e.b) { Right(it) }
    }
```

What does `codistribute` mean for `Gen`? If we have either a generator for `A` or a generator for `B`, we can construct a generator that produces either `A` or `B` depending on which generator we

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

actually have.

We just came up with two general and potentially useful combinators based purely on the abstract interface of `Functor`, and we can reuse them for any type that allows an implementation of `map`.

11.1.1 The importance of laws and their relation to functor

Whenever we create an abstraction like `Functor`, we should not only consider what abstract methods it should have, but also the *laws* we expect it to hold for the implementations. The laws you stipulate for an abstraction are entirely up to you, although Kotlin won't enforce any of these laws on your behalf. If you are going to borrow the name of some existing mathematical abstraction like *functor* or *monoid*, we recommend using the laws already specified by mathematics. Laws are important for two reasons:

- Laws help an interface form a new semantic level whose algebra may be reasoned about *independently* of the instances. For example, when we take the product of a `Monoid<A>` and a `Monoid<B>` to form a `Monoid<Pair<A,B>>`, the monoid laws let us conclude that the "fused" monoid operation is also associative. We don't need to know anything about `A` and `B` to conclude this.
- On a concrete level, we often rely on laws when writing various combinators derived from the functions of some abstract interface like `Functor`. We'll see examples of this later in this section.

For `Functor`, we'll stipulate the familiar law we first introduced in chapter 7 for our `Par` data type. This laws stipulated that relation between the `map` combinator and an identity function as follows:

Listing 11.2 The functor law showing the relation between `map` and identity function

```
map(x) { a -> a } == x
```

In other words, mapping over a structure `x` with the identity function should itself be an identity. This law seems quite natural, and as we progressed beyond `Par`, we noticed that this law was satisfied by the `map` functions of other types like `Gen` and `Parser` too. This law captures the requirement that `map(x)` "preserves the structure" of `x`. Implementations satisfying this law are restricted from doing strange things like throwing exceptions, removing the first element of a `List`, converting a `Some` to `None`, and so on. Only the *elements* of the structure are modified by `map`; the shape or structure itself is left intact. Note that this law holds for `List`, `Option`, `Par`, `Gen`, and most other data types that define `map`!

To give a concrete example of this preservation of structure, we can consider `distribute` and `codistribute`, defined earlier. Here are their signatures again for reference:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A, B> distribute(fab: Kind<F, Pair<A, B>>): Pair<Kind<F, A>, Kind<F, B>>

fun <A, B> codistribute(e: Either<Kind<F, A>, Kind<F, B>>): Kind<F, Either<A, B>>
```

Since we know nothing about `F` other than that it is a functor, the law assures us that the returned values will have the same *shape* as the arguments. If the input to `distribute` is a list of pairs, the returned pair of lists will be of the same length as the input, and corresponding elements will appear in the same order. This kind of algebraic reasoning can potentially save us a lot of work, since reliance on this law means we don't have to write separate tests for these properties.

11.2 Monads: generalizing the `flatMap` and `unit` functions

Now that we understand a bit more about `Functor` and how to apply it, we discover that like `Monoid`, the `Functor` is just one of many abstractions that can be factored out of our libraries. But `Functor` isn't the most compelling abstraction, as there aren't that many useful operations that can be defined purely in terms of `map`.

Instead, let's focus our attention on the more interesting interface called `Monad` that *adds* to the functionality of `Functor`. Using this interface, we can implement far more operations than with the functor alone, all while factoring out what would otherwise be duplicate code. It also comes with laws that allow us to reason about how our libraries behave in the way that we expect them to.

Recall that for several of the data types in this book, we have implemented `map2` to "lift" a function taking two parameters. For `Gen`, `Parser`, and `Option`, the `map2` function could be implemented as follows.

Listing 11.3 Implementations of `map2` for `Gen`, `Parser` and `Option`

```
fun <A, B, C> map2(
    fa: Gen<A>,
    fb: Gen<B>,
    f: (A, B) -> C
): Gen<C> =  ❶
    flatMap(fa) { a -> map(fb) { b -> f(a, b) } }

fun <A, B, C> map2(
    fa: Parser<A>,
    fb: Parser<B>,
    f: (A, B) -> C
): Parser<C> =  ❷
    flatMap(fa) { a -> map(fb) { b -> f(a, b) } }

fun <A, B, C> map2(
    fa: Option<A>,
    fb: Option<B>,
    f: (A, B) -> C
): Option<C> =  ❸
    flatMap(fa) { a -> map(fb) { b -> f(a, b) } }
```

❶ Make a generator of a random `C` that runs random generators `fa` and `fb`, combining their results with the function `f`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❷ Make a parser that produces `C` by combining the results of parsers `fa` and `fb` with the function `f`.

❸ Combines two `Options` with the function `f` if both have a value, otherwise returns `None`.

These functions have more in common than just the name. In spite of operating on data types that seemingly have nothing to do with one another, the implementations are identical! The only thing that differs is the particular data type being operated on. This confirms what we've been hinting at all along—that these are particular instances of some more general pattern. We should be able to exploit such a pattern to avoid repeating ourselves. For example, we should be able to write `map2` only once in such a way that it can be reused for all of these data types.

We've made the code duplication particularly obvious here by choosing uniform names for our functions and parameters, taking the arguments in the same order, and so on. It may be bit more difficult to spot in your everyday work. But the more libraries you write, the better you'll get at identifying patterns that you can factor out into common abstractions.

11.2.1 Introducing the `Monad` interface

Monads are everywhere! In fact, this is what unites `Parser`, `Gen`, `Par`, `Option`, and many of the other data types we've looked at so far. Much like we did with `Foldable` and `Functor`, we can come up with a Kotlin interface for `Monad` that defines `map2` and numerous other functions once and for all, rather than having to duplicate their definitions for every concrete data type.

In part 2 of this book, we concerned ourselves with individual data types, finding a minimal set of primitive operations from which we could derive a large number of useful combinators. We'll do the same kind of thing here to refine an *abstract* interface to a small set of primitives.

Let's start by introducing a new interface, call it `Mon` for now. Since we know that we eventually want to define `map2`, let's go ahead and define it as in listing 11.4.

Listing 11.4 Defining a `Mon` interface as home for `map2`

```
interface Mon<F> {   ❶

    fun <A, B, C> map2(
        fa: Kind<F, A>,   ❷
        fb: Kind<F, B>,   ❸
        f: (A, B) -> C
    ): Kind<F, C> =
        flatMap(fa) { a -> map(fb) { b -> f(a, b) } }   ❹
}
```

❶ The `Mon` interface is parameterized with higher-kinded type of `F`

❷ Use `Kind<F, A>` to represent `F<A>`

❸ Will not compile since `map` and `flatMap` are not defined in context of `F`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

In this sample, we've just taken the implementation of `map2` and changed `Parser`, `Gen`, and `Option` to the polymorphic `F` of the `Mon<F>` interface in the signature.[52] We refer to in-place references to the kind of `F` using the `Kind` interface. But in this polymorphic context, the implementation won't compile! We don't know *anything* about `F` here, so we certainly don't know how to `flatMap` or `map` over an `Kind<F, A>`!

What we can do is simply *add* `map` and `flatMap` to the `Mon` interface and keep them abstract. In doing so, we keep `map2` consistent with what we had before.

Listing 11.5 Introducing `flatMap` and `map` declarations to the `Mon` interface

```
fun <A, B> map(fa: Kind<F, A>, f: (A) -> B): Kind<F, B>

fun <A, B> flatMap(fa: Kind<F, A>, f: (A) -> Kind<F, B>): Kind<F, B>
```

This translation was rather mechanical. We just inspected the implementation of `map2`, and added all the functions it called, `map` and `flatMap`, as suitably abstract methods on our interface. This interface will now compile, but before we declare victory and move on to defining instances of `Mon<List>`, `Mon<Parser>`, `Mon<Option>` and so on, let's see if we can refine our set of primitives. Our current set of primitives is `map` and `flatMap`, from which we can derive `map2`. Is `flatMap` and `map` a minimal set of primitives? Well, the data types that implemented `map2` all had a `unit`, and we know that `map` can be implemented in terms of `flatMap` and `unit`. For example, on `Gen`:

```
fun <A, B> map(fa: Gen<A>, f: (A) -> B): Gen<B> =
    flatMap(fa) { a -> unit(f(a)) }
```

So let's pick `flatMap` and `unit` as our minimal set of primitives. We'll unify all data types under a single concept that have these functions defined. The interface shall be called `Monad`, and has abstract declarations of `flatMap` and `unit`, while providing default implementations for `map` and `map2` in terms of our abstract primitives.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 11.6 Declaration of the `Monad` with primitives defined for `flatMap` and `unit`.

```
interface Monad<F> : Functor<F> {   ❶

    fun <A> unit(a: A): Kind<F, A>

    fun <A, B> flatMap(fa: Kind<F, A>, f: (A) -> Kind<F, B>): Kind<F, B>

    override fun <A, B> map(
        fa: Kind<F, A>,
        f: (A) -> B
    ): Kind<F, B> =   ❷
        flatMap(fa) { a -> unit(f(a)) }

    fun <A, B, C> map2(
        fa: Kind<F, A>,
        fb: Kind<F, B>,
        f: (A, B) -> C
    ): Kind<F, C> =
        flatMap(fa) { a -> map(fb) { b -> f(a, b) } }
}
```

❶ `Monad` provides a default implementation of `map` and can so implement `Functor`

❷ The override of `map` in `Functor` needs to be made explicit for successful compilation

SIDEBAR **What the monad name means**

We could have called `Monad` anything at all, like `FlatMappable`, `Unicorn`, or `Bicycle`. But *monad* is already a perfectly good name in common use. The name comes from category theory, a branch of mathematics that has inspired a lot of functional programming concepts. The name *monad* is intentionally similar to *monoid*, and the two concepts are related in a deep way.

To tie this back to a concrete data type, we can implement the `Monad` instance for `Gen`.

Listing 11.7 Declaring a `Monad` instance for `Gen` using concrete types

```
object Monads {

    val genMonad = object : Monad<ForGen> {   ❶

        override fun <A> unit(a: A): GenOf<A> = Gen.unit(a)   ❶

        override fun <A, B> flatMap(
            fa: GenOf<A>,
            f: (A) -> GenOf<B>
        ): GenOf<B> =
            fa.fix().flatMap { a: A -> f(a).fix() }   ❸
    }
}
```

❶ The type `ForGen` is a surrogate type we provide to get around Kotlin's limitations in expressing higher-kinded types

❷ The type alias `GenOf<A>` is syntactic sugar for `Kind<ForGen, A>`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❸ Down-cast all `GenOf<A>` to `Gen<A>` using provided extenstion method `fix()` for compatibility with `Gen.flatMap`

We only need to implement `flatMap` and `unit`, and we get `map` and `map2` at no additional cost. This is because `Monad` inherits these two functions from `Functor`. We've implemented them once and for all, and this for any data type which allows an instance of `Monad` to be created! But we're just getting started. There are many more such functions that we can implement in this manner.

EXERCISE 11.1

Write monad instances for `Par`, `Option` and `List`. Additionally, provide monad instances for `arrow.core.ListK` and `arrow.core.SequenceK`.

NOTE

The `ListK` and `SequenceK` types provided by Arrow are wrapper classes that turn their platform equivalents, `List` and `Sequence`, into fully equipped type constructors.

EXERCISE

```
object Monads {

    val parMonad: Monad<ForPar> = TODO()

    val optionMonad: Monad<ForOption> = TODO()

    val listMonad: Monad<ForList> = TODO()

    val listKMonad: Monad<ForListK> = TODO()

    val sequenceKMonad: Monad<ForSequenceK> = TODO()
}
```

EXERCISE 11.2 (Hard)

`State` looks like it could be a monad too, but it takes two type arguments, namely `S` and `A`. You need a type constructor of only one argument to implement `Monad`. Try to implement a `State` monad, see what issues you run into, and think about if or how you can solve this. We'll discuss the solution later in this chapter.

```
data class State<S, out A>(val run: (S) -> Pair<A, S>) : StateOf<S, A>
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

11.3 Monadic combinators

We have already come to a point where we've defined primitives for the monad. Equipped with these, we can now move ahead and discover additional combinators. In fact, now we can look back at previous chapters and see if there were some other functions that we implemented for each of our monadic data types. Many of these types can be implemented as once-for-all monads, so let's do that now.

EXERCISE 11.3

The `sequence` and `traverse` combinators should be pretty familiar to you by now, and your implementations of them from previous chapters are probably all very similar. Implement them once and for all on `Monad<F>`.

```
fun <A> sequence(lfa: List<Kind<F, A>>): Kind<F, List<A>> = TODO()

fun <A, B> traverse(
    la: List<A>,
    f: (A) -> Kind<F, B>
): Kind<F, List<B>> = TODO()
```

One combinator we saw for `Gen` and `Parser` was `listOfN`, which allowed us to replicate a generator or parser `n` times to get a parser or generator of lists of that length. We can implement this combinator for all monads `F` by adding it to our `Monad` interface. We could also give it a more generic name such as `replicateM`, meaning "replicate in a monad".

EXERCISE 11.4

Implement `replicateM` to generate a `Kind<F, List<A>>`, with the list being of length `n`.

```
fun <A> replicateM(n: Int, ma: Kind<F, A>): Kind<F, List<A>> = TODO()

fun <A> _replicateM(n: Int, ma: Kind<F, A>): Kind<F, List<A>> = TODO()
```

EXERCISE 11.5

Think about how `replicateM` will behave for various choices of `F`. For example, how does it behave in the `List` monad? And what about `Option`? Describe in your own words the general meaning of `replicateM`.

There was also a combinator for our `Parser` data type called `product`, which took two parsers and turned them into a parser of pairs. We implemented this `product` combinator in terms of `map2`. We can also write it generically for any monad `F`:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 11.8 Generic implementation of `product` using the `map2` combinator.

```
fun <A, B> product(
    ma: Kind<F, A>,
    mb: Kind<F, B>
): Kind<F, Pair<A, B>> =
    map2(ma, mb) { a, b -> Pair(a, b) }
```

We don't have to restrict ourselves to combinators that we've seen already. We should take the liberty to explore new solutions too.

EXERCISE 11.6 (Hard)

Here's an example of a function we haven't seen before. Implement the function `filterM`. It's a bit like `filter`, except that instead of a function from `(A) -> Boolean`, we have an `(A) -> Kind<F, Boolean>`. Replacing various ordinary functions like `filter` with the monadic equivalent often yields interesting results. Implement this function, and then think about what it means for various data types such as `Par`, `Option` and `Gen`.

```
fun <A> filterM(
    ms: List<A>,
    f: (A) -> Kind<F, Boolean>
): Kind<F, List<A>> = TODO()
```

The combinators we've seen here are only a small sample of the full library that `Monad` lets us implement once and for all. We'll see some more examples in chapter 13.

11.4 Monad laws

Algebraic concepts like monad and functor are embodiments of the laws that define and govern them. In this section, we'll introduce the laws that govern our `Monad` interface. Certainly we'd expect the functor laws to also hold for `Monad`, since a `Monad<F>` *is* a `Functor<F>`, but what else do we expect? What laws should constrain `flatMap` and `unit`? In short, we can cite several laws that fulfill these constraints:

- the associative law
- the *left* identity law
- the *right* identity law

This section will look at each one in turn, all while proving that they hold for the monad.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

11.4.1 The associative law

The first monadic law that we will look into is the *associative* law. This law is all about ordering of operations. Let's look at this by way of example. If we wanted to combine three monadic values into one, which two should we combine first? Should it matter? To answer this question, for a moment let's step away from the abstract level and look at a simple concrete example using the `Gen` monad.

Say that we're testing a product order system and we need to generate some fake orders as fixture for our test. We might have an `Order` data class and a generator for that class.

Listing 11.9 Declaration for an `Item` and `Order` text fixture generator

```
data class Order(val item: Item, val quantity: Int)
data class Item(val name: String, val price: Double)

val genOrder: Gen<Order> =
    Gen.string().flatMap { name: String ->   ❶
        Gen.double(0..10).flatMap { price: Double ->   ❶
            Gen.choose(1, 100).map { quantity: Int ->   ❷
                Order(Item(name, price), quantity)
            }
        }
    }
```

❶ Generate a random string `name`

❷ Generate a double `price` between 0 and 10

❸ Generate an integer `quantity` between 1 and 100

Here we're generating the `Item` inline (from `name` and `price`), but there might be places where we want to generate an `Item` separately. We could pull that into its own generator:

```
val genItem: Gen<Item> =
    Gen.string().flatMap { name: String ->
        Gen.double(0..10).map { price: Double ->
            Item(name, price)
        }
    }
```

This can now in turn can be used to generate orders:

```
val genOrder2: Gen<Order> =
    Gen.choose(1, 100).flatMap { quantity: Int ->
        genItem.map { item: Item ->
            Order(item, quantity)
        }
    }
```

And that should do exactly the same thing, right? It seems safe to assume that. But not so fast! How can we be sure? It's not exactly the same code!

Let's expand the implementation of `genOrder` into calls to `map` and `flatMap` to better see what's

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

going on.

```
val genOrder3: Gen<Order> =
    Gen.choose(1, 100).flatMap { quantity: Int ->
        Gen.string().flatMap { name: String ->
            Gen.double(0..10).map { price: Double ->
                Order(Item(name, price), quantity)
            }
        }
    }
```

When we compare this with listing 11.8 we can clearly see that they are *not* identical, yet it seems perfectly reasonable to assume that the two implementations do exactly the same thing. In fact, even though the order has changed, it would be surprising and weird if they didn't. It's because we're assuming that `flatMap` obeys an *associative law*.

Listing 11.10 Expressing the law of associativity in terms of `flatMap`

```
x.flatMap(f).flatMap(g) ==
    x.flatMap { a -> f(a).flatMap(g) }
```

And this law should hold for all values `x`, `f` and `g` of the appropriate types—not just for `Gen` but for `Parser`, `Option`, or any other monad.

11.4.2 Proving the associative law for a specific monad

Up to this point, we've been dealing strictly at an abstract level. But what bearing does this have on a real-world situation. How does this apply to the data types we have dealt with in past chapters? To find out, let's *prove* that this law holds for `Option`. All we have to do is substitute `None` or `Some(v)` for `x` in the preceding equation and expand both it's sides. We will start with the case where `x` is `None`:

```
None.flatMap(f).flatMap(g) ==
    None.flatMap { a -> f(a).flatMap(g) }
```

Since `None.flatMap(f)` is `None` for all `f`, this can be simplified to:

```
None == None
```

In other words, the law holds for `None`. Let's confirm that the same is true when `x` is `Some(v)` for an arbitrary value `v`.

Listing 11.11 Verifying the associative law by substitution of `x` by `Some(v)`

```
x.flatMap(f).flatMap(g) == x.flatMap { a -> f(a).flatMap(g) }   ❶

Some(v).flatMap(f).flatMap(g) ==
    Some(v).flatMap { a -> f(a).flatMap(g) }   ❷

f(v).flatMap(g) == { a: Int -> f(a).flatMap(g) }(v)   ❸

f(v).flatMap(g) == f(v).flatMap(g)   ❹
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

1. Original law of associativity for `flatMap`
2. Substitute `x` with `Some(v)` on both sides
3. Collapse `Some(v).flatMap` on both sides by applying `v` to `f` directly
4. Apply `v` to `g` directly on the right side, proving equality

Thus we can conclude, this law also holds when `x` is `Some(v)` for any value of `v`. We can so conclude that the law holds for both cases of `Option`.

KLEISLI COMPOSITION: A CLEARER VIEW ON THE ASSOCIATIVE LAW

It's not so easy to recognize the law of associativity in the preceding example. In contrast, remember how clear the associative law for monoids was?

```
combine(combine(x,y), z) == combine(x, combine(y,z))
```

Our associative law for monads looks nothing like that! Fortunately for us, there is a way to make this law clearer by considering monadic *functions* instead of monadic values as we have been doing up to now.

What exactly do we mean by a monadic function, and how does it differ from the monadic values we have seen so far? If a monadic value is an instance of `F<A>`, a monadic function is a function in the form of `(A)  F<B>`. A function such as this is known as a *Kleisli arrow*, and is named after the Swiss mathematician Heinrich Kleisli. What makes Kleisli arrows special is that they can be *composed* with each other:

```
fun <A, B, C> compose(
    f: (A) -> Kind<F, B>,
    g: (B) -> Kind<F, C>
): (A) -> Kind<F, C>
```

EXERCISE 11.7

Implement the following Kleisli composition function in `Monad`.

```
fun <A, B, C> compose(
    f: (A) -> Kind<F, B>,
    g: (B) -> Kind<F, C>
): (A) -> Kind<F, C> = TODO()
```

Considering that `flatMap` takes a Kleisli arrow as parameter, we can now state the associative law for monads using this new function in a far more symmetric way.

Listing 11.12 Expressing the law of associativity in terms of `compose`

```
compose(compose(f, g), h) == compose(f, compose(g, h))
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 11.8 (Hard)

Implement `flatMap` in terms of an abstract definition of `compose`. By this, it seems as though we've found another minimal set of monad combinators: `compose` and `unit`.

```
fun <A, B> flatMap(
    fa: Kind<F, A>,
    f: (A) -> Kind<F, B>
): Kind<F, B> = TODO()
```

VERIFYING ASSOCIATIVITY IN TERMS OF FLATMAP AND COMPOSE

In listing 11.10, we expressed the associative law for monads in terms of `flatMap`. We then chose a clearer representation of this law using `compose` in listing 11.12. In this section, we will prove that the two proofs are equivalent by applying the substitution model to the law expressed in terms of `compose` using the implementation in terms of `flatMap` derived in exercise 11.8. We will look at one side at a time for the sake of simplicity. Let's focus on the left side of the equation first.

Listing 11.13 Apply the substitution model to left side of associative law in terms of `compose`

```
compose(compose(f, g), h)   ❶

{ a -> flatMap(compose(f, g)(a), h) }   ❷

{ a -> flatMap({ b: A -> flatMap(f(b), g) }(a), h) }   ❸

{ a -> flatMap(flatMap(f(a), g), h) }   ❹

flatMap(flatMap(x, g), h)   ❺
```

❶ Left side of law of associativity expressed in terms of `compose`

❷ Substitute outer `compose` with `flatMap`, propagating `a`

❸ Substitute inner `compose` with `flatMap`, propagating `b`

❹ Apply `a` through `b` to `f`

❺ Simplify by introducing alias `x` for any `f` with `a` applied

Next, we shift our attention to the right-hand side.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 11.14 Apply substitution model to right side of associative law in terms of `compose`

```
compose(f, compose(g, h))   ❶

{ a -> flatMap(f(a), compose(g, h)) }   ❷

{ a -> flatMap(f(a)) { b -> flatMap(g(b), h) } }   ❸

flatMap(x) { b -> flatMap(g(b), h) }   ❹
```

❶ Right side of law of associativity expressed in terms of `compose`

❷ Substitute outer `compose` with `flatMap`, propagating `a`

❸ Substitute inner `compose` with `flatMap`, propagating `b`

❹ Simplify by introducing alias `x` for any `f` with `a` applied

The final outcome looks like this:

```
flatMap(flatMap(x, g), h) ==
    flatMap(x) { b -> flatMap(g(b), h) }
```

We can express this more simply by making `flatMap` an extension function on the higher kind `x`.

```
x.flatMap(g).flatMap(h) ==
    x.flatMap { b -> g(b).flatMap(h) }
```

Apart from the naming of some of the parameters, this aligns perfectly with the law stated in terms of `flatMap` in listing 11.10. We can thus conclude that the proofs are equivalent.

11.4.3 The left and right identity laws

The other laws used to define the monad are called the *identity* laws. It is worth mentioning that this is not just a single law but a *pair* of laws, referred to as *left identity* and *right identity*. Collectively with the associative law, they're often referred to as the *three monad laws*.

Let's begin by thinking about what *identity* means. Just like `nil` was an *identity element* for `combine` in monoid, there is also an identity element for `compose` in monad. The name *unit* is often used in mathematics to mean an *identity* for some operation, so it goes to follow that we chose `unit` for the name of our monad identity operation.

```
fun <A> unit(a: A): Kind<F, A>
```

Now that we have a way of defining the identity element, we will use it in conjunction with `compose` to express the two identity laws. Recall from exercise 11.7 that `compose` takes two arguments, one of type `(A) Kind<F, B>` and the other of `(B) Kind<F, C>`. The `unit`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

function has the right type to be passed as an argument to `compose`. The effect should be that anything composed with `unit` is that same thing. This usually takes the form of our two laws, *left identity* and *right identity*:

```
compose(f, { a: A -> unit(a) }) == f
compose({ a: A -> unit(a) }, f) == f
```

We can also state these laws in terms of `flatMap`, but they're less clear to understand that way:

```
flatMap(x) { a -> unit(a) } == x
flatMap(unit(a), f) == f(a)
```

EXERCISE 11.9

Using the following values, prove that the left and right identity laws expressed in terms of `compose` are equivalent to that stated in terms of `flatMap`:

```
val f: (A) -> Kind<F, A>
val x: Kind<F, A>
val v: A
```

EXERCISE 11.10

Prove that the identity laws hold for the `Option` monad.

EXERCISE 11.11

Monadic combinators can be expressed in another minimal set, namely `map`, `unit`, and `join`. Implement the `join` combinator in terms of `flatMap`.

EXERCISE 11.12

Either `flatMap` or `compose` may now be implemented in terms of `join`. For the sake of this exercise, implement both.

EXERCISE 11.13 (Hard/Optional)

Restate the monad law of associativity in terms of `flatMap` using `join`, `map` and `unit`.

EXERCISE 11.14 (Hard/Optional)

In your own words, write down an explanation of what the associative law means for `Par` and `Parser`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 11.15 (Hard/Optional)

Explain in your own words what the identity laws are stating in concrete terms for `Gen` and `List`.

Recall the identity laws for left and right identity respectively:

```
flatMap(x) { a -> unit(a) } == x
flatMap(unit(a), f) == f(a)
```

11.5 Just what is a monad?

Up to now, we've been examining monads at the micro-level by identifying various combinators and proving associated laws. Even though this is useful, it doesn't really tell us much about what a monad is. In order to further our understanding, we will zoom out to a broader perspective on this subject. In doing so, we see something unusual about the `Monad` interface. The data types for which we've given monad instances don't seem to have much to do with each other. Yes, `Monad` factors out code duplication among them, but what *is* a monad exactly? What does "monad" *mean*?

You may be used to thinking of interfaces as providing a relatively complete API for an abstract data type, merely abstracting over the specific representation. After all, a singly linked list and an array-based list may be implemented differently behind the scenes, but they'll probably share a common `List` interface in terms of which a lot of useful and concrete application code can be written. `Monad`, like `Monoid`, is a more abstract and purely algebraic interface. The `Monad` combinators are often just a small fragment of the full API for a given data type that happens to be a monad. So `Monad` doesn't generalize one type or another; rather, many vastly different data types can satisfy the `Monad` interface and laws.

We've seen three minimal sets of primitive monadic combinators, and instances of `Monad` will have to provide implementations of one of these sets:

- `flatMap` and `unit`
- `compose` and `unit`
- `map`, `join` and `unit`

We also know that there are two monad laws to be satisfied, namely *associativity* and *identity*, that can be formulated in various ways. So we can state plainly what a monad *is*:

> *A monad is an implementation of one of the minimal sets of monadic combinators, satisfying the laws of associativity and identity.*

That's a perfectly respectable, precise and terse definition. And if we're being precise, this is the

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

only correct definition. A monad is defined by its operations and laws; no more, no less. But it's a little unsatisfying. It doesn't say much about what it implies—what a monad *means*. The problem is that it's a *self-contained* definition. Even if you're an experienced programmer that has obtained a vast amount of knowledge related to programming, this definition does not intersect with any of that accumulated knowledge.

11.5.1 The identity monad

In order to really *understand* what's going on with monads, let's try to think about them in terms of things we already know, and then we will connect them to a wider context. To develop some intuition for what monads *mean*, let's look at some more monads and compare their behavior.

To distill monads to their most essential form, we look to the simplest interesting specimen, the identity monad, given by the following type:

```
data class Id<A>(val a: A)
```

EXERCISE 11.16

Implement `map`, `flatMap` and `unit` as methods on this class, and give an implementation for `Monad<Id>`.

```
data class Id<out A>(val a: A) : IdOf<A> {
    companion object {
        fun <A> unit(a: A): Id<A> = TODO()
    }

    fun <B> flatMap(f: (A) -> Id<B>): Id<B> = TODO()
    fun <B> map(f: (A) -> B): Id<B> = TODO()
}

val idMonad: Monad<ForId> = TODO()
```

Now, `Id` is just a simple wrapper. It doesn't really add anything. Applying `Id` to `A` is an identity since the wrapped type and the unwrapped type are totally isomorphic (we can go from one to the other and back again without any loss of information). But what is the meaning of the identity *monad*? Let's try using it in some code:

```
val id: Id<String> = idMonad.flatMap(Id("Hello, ")) { a: String ->
    idMonad.flatMap(Id("monad!")) { b: String ->
        Id(a + b)
    }
}.fix()
```

When evaluating `id` in the REPL, we find the following result:

```
>>> id
res1: example.Id(a=Hello, monad!)
```

So what is the *action* of `flatMap` for the identity monad in the example? It's simply variable

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

substitution. The variables `a` and `b` get bound to `"Hello, "` and `"monad!"` respectively, and then substituted into the expression `a + b`. We could have written the same thing without the `Id` wrapper using simple variables:

```
>>> val a = "Hello, "
>>> val b = "monad!"
>>> a + b
res2: kotlin.String = Hello, monad!
```

Besides the `Id` wrapper, there is no difference. So now we have at least a partial answer to the question of what a monad means. We could say that monads provide a context for introducing and binding variables, and allowing variable substitution. But is there more to it than that?

11.5.2 The `State` monad and partial type application

We have examined the simplest possible case by observing the `Id` monad in the previous section. We will now shift our focus to the opposite end of the spectrum by way of looking at a more challenging monad that we dealt with in chapter 6, the `State` monad.

If you recall this data type, you will remember that we wrote `flatMap` and `map` functions in exercise 6.8 and 6.9 respectively. Let's take another look at this data type with its combinators.

Listing 11.15 The `State` data type as previously established for representing state transitions

```
data class State<S, out A>(val run: (S) -> Pair<A, S>) {

    companion object {
        fun <S, A> unit(a: A): State<S, A> =
            State { s: S -> Pair(a, s) }

    }

    fun <B> map(f: (A) -> B): State<S, B> =
        flatMap { a -> unit<S, B>(f(a)) }

    fun <B> flatMap(f: (A) -> State<S, B>): State<S, B> =
        State { s: S ->
            val (a: A, s2: S) = this.run(s)
            f(a).run(s2)
        }
}
```

It seems like `State` definitely fits the profile of being a monad, but there does seem to be a caveat. If you had a play with this in exercise 11.2, you would have noticed the problem is that the type constructor takes *two* type arguments, while `Monad` requires a type constructor of only one. This means we can't simply get away with declaring `Monad<ForState>` as the surrogate type `ForState` would need to imply a `State<S, A>`. This having two type parameters, not one.

If we choose some particular `S`, then we have something like `ForStateS` and `StateOfS<A>`, which is closer to the kind expected by `Monad`. In other words, `State` doesn't have a single

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

monad instance, but a *whole family* of them, one for each choice of `S`. What we really want to do is to *partially apply* `State` to where the `S` type argument is fixed to be some concrete type, resulting in only one remaining type variable, `A`.

This is much like how we would partially apply a function, except now we do it at the type level. For example, we can create an `IntState` type constructor, which is an alias for `State` with its first type argument fixed to be `Int`:

```
typealias IntState<A> = State<Int, A>
```

And `IntState` is exactly the kind of thing that we can build a `Monad` for:

Listing 11.16 A state monad instance partially applied for `Int` types

```
val intState = object : Monad<ForIntState> {        ❶
    override fun <A> unit(a: A): IntStateOf<A> =    ❷
        IntState { s: Int -> Pair(a, s) }

    override fun <A, B> flatMap(
        fa: IntStateOf<A>,
        f: (A) -> IntStateOf<B>
    ): IntStateOf<B> =
        fa.fix().flatMap { a: A -> f(a).fix() }
}
```

❶ A surrogate type in substitution of `Kind<Int, A>` to appease the compiler

❷ A type alias for `Kind<ForIntState, A>`

Of course, this would be really repetitive if we had to write an explicit `Monad` instance for every specific state type. Consider `IntState`, `DoubleState`, `StringState`, the list goes on. Beside the fact that this doesn't scale well, it would also mean that our `State` data type would need to inherit from `IntState`, along with every other partially applied type in the family of monads. This simply isn't possible in Kotlin!

Putting this approach of hardcoded monad instances aside, let's look at how we can solve this with less code duplication. Fortunately there is a way of doing this by introducing the `StateMonad` interface that can be partially applied with a type such as `Int`, resulting in a `StateMonad<Int>`.

Listing 11.17 The state monad interface does away with hardcoded partially applied types

```
interface StateMonad<S> : Monad<StatePartialOf<S>> {   ❶

    override fun <A> unit(a: A): StateOf<S, A>   ❷

    override fun <A, B> flatMap(
        fa: StateOf<S, A>,
        f: (A) -> StateOf<S, B>
    ): StateOf<S, B>
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❶ The `Monad` type constructor takes partially applied type parameter `StatePartialOf` for any `S`

❷ Monadic combinators no longer restricted to deal in single type parameter currency

The main difference comes in the declaration of the `StateMonad` interface itself. The monad interface has a type parameter `S` for the family member that it represents, and it will extend from a new type alias `StatePartialOf<S>`, that is an alias for `Kind<ForState, S>`.

The types such as `StatePartialOf` and `StateOf` are merely boilerplate code that we can write ourselves, although Arrow already conveniently generates all this for us. Appendix D of the book describes exactly what boilerplate is required, as well as how to let Arrow do all the hard work on our behalf.

NOTE Please revise appendix D section 2 that describes the boilerplate code required for declaring a partially applied type constructor. **Please ensure that you have read and thoroughly understood this content.**

We can now declare a new member of the state monad family using the `StateMonad` interface. Let's stick with our `intStateMonad` example from before using this interface and boilerplate code:

Listing 11.18 A partially applied `State` monad that brings flexibility to the type family members it represents

```
val intStateMonad: StateMonad<Int> = object : StateMonad<Int> {
    override fun <A> unit(a: A): StateOf<Int, A> =
        State { s -> Pair(a, s) }

    override fun <A, B> flatMap(
        fa: StateOf<Int, A>,
        f: (A) -> StateOf<Int, B>
    ): StateOf<Int, B> =
        fa.fix().flatMap { a -> f(a).fix() }
}
```

We can see that we have evolved from the hard-coded `ForIntState` monad in listing 11.16 to a more flexible partially applied variant in listing 11.17. Once more, just by giving implementations of `flatMap` and `unit`, we get implementations of all the other monadic combinators for free.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 11.17

Now that we have a `State` monad, try it out to see how it behaves. Declare some values of `replicateM`, `map2` and `sequence` with type declarations using the `intMonad` above. Describe how each one behaves under the covers?

```
val replicateIntState: StateOf<Int, List<Int>> = TODO()

val map2IntState: StateOf<Int, Int> = TODO()

val sequenceIntState: StateOf<Int, List<Int>> = TODO()
```

Now that we've examined both `Id` and `State`, we can once again take a step back ask what the meaning of *monad* is. Let's look at the difference between the two. Remember from chapter 6 that the primitive operations on `State` (besides the monadic operations `flatMap` and `unit`) are that we can modify the current state through the use of some form of get and set combinators:

```
fun <S> getState(): State<S, S> = State { s -> Pair(s, s) }

fun <S> setState(s: S): State<S, Unit> = State { Pair(Unit, s) }
```

Remember that we also discovered that these combinators constitute a minimal set of primitive operations for `State`. So together with the monadic primitives, `flatMap` and `unit`, they *completely specify* everything that we can do with the `State` data type. This is true in general for monads—they all have `flatMap` and `unit`, and each monad brings its own set of additional primitive operations that are specific to that monad.

EXERCISE 11.18

Express the laws that you would expect to mutually hold for `getState`, `setState`, `flatMap`, and `unit`?

What does this tell us about the meaning of the `State` *monad* ? To fully grasp what we're trying to convey, let's once again turn our attention to the `intStateMonad` from listing 11.18 by using it in a real example.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Listing 11.19 Getting and setting state with sequence of `flatMap` and `map` operations

```
val F = intStateMonad

fun <A> zipWithIndex(la: List<A>): List<Pair<Int, A>> =
    la.foldLeft(F.unit(emptyList<Pair<Int, A>>())) { acc, a ->
        acc.fix().flatMap { xs ->
            acc.fix().getState<Int>().flatMap { n ->
                acc.fix().setState(n + 1).map { _ ->
                    listOf(Pair(n, a)) + xs
                }
            }
        }
    }.fix().run(0).first.reversed()
```

This function numbers all the elements in a list using a `State` action. It keeps a state that's an `Int`, which is incremented at each step. We run the composite `State` action starting from `0`. Finally we reverse the order since we ran the computation in reverse using `foldLeft`.

To express this even clearer, we can imagine the body passed to the `leftFold` using an Arrow-style for-comprehension in this snippet of pseudo-code:

Listing 11.20 Getting and setting state with a for-comprehension

```
...
{ acc: StateOf<Int, List<Pair<Int, A>>>, a: A ->
    acc.fx {
        val (xs) = acc
        val (n) = acc.getState()
        val (_) = acc.setState(n + 1)
        listOf(Pair(n, a)) + xs
    }
}
...
```

The for-comprehension removes all the clutter introduced by `flatMap` and `map` and let's us focus on what seems like a sequence of imperative instructions using the `State` to propagate an incrementing counter.

Note what's going on with `getState` and `setState` in the for-comprehension. We're obviously getting variable binding just like in the `Id` monad—we're binding the value of each successive state action (`acc`, `getState`, and then `setState`) to variables. But there's more going on here *between the lines*. At each line in the for—comprehension, the implementation of `flatMap` is making sure that the current state is available to `getState`, and that the new state gets propagated to all actions that follow a `setState`.

What does the difference between the action of `Id` and the action of `State` tell us about monads in general? We can see that a chain of `flatMap` calls (or an equivalent for-comprehension) is like an imperative program with statements that assign to variables, and the *monad specifies what occurs at statement boundaries*. For example, with `Id`, nothing at all occurs except unwrapping

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

and rewrapping in the `Id` constructor. With `State`, the most current state gets passed from one statement to the next. With the `Option` monad, a statement may return `None` and terminate the program. With the `List` monad, a statement may return many results, which causes statements that follow it to potentially run multiple times, once for each result.

The `Monad` contract doesn't specify *what* is happening between the lines, only that whatever *is* happening satisfies the laws of associativity and identity.

EXERCISE 11.19 (Hard)

To cement your understanding of monads, give a monad instance for the `Reader` data type and explain what it means. Also, take some time to answer the following questions:

- what are its primitive operations?
- What is the action of `flatMap`?
- What meaning does it give to monadic functions like `sequence`, `join`, and `replicateM`?
- What meaning does it give to the monadic laws?

In this chapter, we took a pattern that we've seen repeated throughout the book and then unified it under a single concept: monad. This allowed us to write a number of combinators once and for all, for many different data types that at first glance don't seem to have anything in common. We discussed *monad laws* that all monads satisfy from various perspectives, then finally developed some insight into what the broad term means.

An abstract topic like this can't be fully understood all at once. It requires an iterative approach where you keep revisiting the topic from different perspectives. When you discover new monads or new applications of them, or see them appear in a new context, you'll inevitably gain new insight. And each time it happens, you might think to yourself, "OK, I thought I understood monads before, but now I *really* get it.". Don't be fooled!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

11.6 Summary

- The type constructor `F` representing types like `List` or `Option` is a functor, and the `Functor<F>` instance proves that this assumption holds true.
- The functor interface has a `map` method, which is a higher order function that applies a transformation to each element of the enclosing kind.
- Laws have importance because they establish the semantics of an interface. This results in an algebra that may be reasoned about *independently* from its instances.
- The functor law stipulates the relationship between `map` and identity functions. It preserves the structure of the enclosing kind and is only concerned with transforming its elements.
- The monad interface *is a functor* that typically has `flatMap` and `unit` primitives. These primitive functions can be used to derive many other useful combinators, including those of the functor.
- The monadic laws constrain the behavior of a monad by enforcing principles of *associativity* and *identity* on its instances.
- The *associative law* deals with ordering, and it guarantees that outcomes will remain the same no matter how `flatMap` operations are nested.
- The identity laws are comprised of *left identity* and *right identity*, each dealing with a situation where the result of `unit` is the subject or object of a `flatMap` expression.
- There are three minimal sets of combinators that can define a monad, namely `unit` combined with either `flatMap`, `compose` or `map` *and* `join`.
- Each monad has a set of basic primitives along with its own set of additional combinators, the interaction of all these combined making the behavior of each monad unique.
- The monad contract doesn’t specify what is happening *between the lines* of a for-comprehension, only that whatever is happening satisfies the monadic laws.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Appendix A. Exercise hints and tips

A.1 Introduction

This appendix contains hints and tips to get you thinking in the right direction for the more challenging exercises in this book. Trivial exercises have been omitted from this appendix, although full solutions for all exercises can be found in Appendix B.

CHAPTER 3. FUNCTIONAL DATA STRUCTURES

EXERCISE 3.1

Implement the function `tail` for removing the first element of a `List`. Note that the function takes constant time. What are different choices you could make in your implementation if the `List` is `Nil`? We'll return to this question in the next chapter.

TIP	Try matching on the list's *element type*. Consider carefully how you would deal with an empty list.

EXERCISE 3.2

Using the same idea, implement the function `setHead` for replacing the first element of a `List` with a different value.

TIP	The same applies here as for Exercise 3.1

EXERCISE 3.3

Generalize `tail` to the function `drop`, which removes the first `n` elements from a list. Note that this function takes time proportional only to the number of elements being dropped—we don't need to make a copy of the entire `List`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP Use matching and recursion to solve this problem. Consider all the following scenarios in your solution:

- What should the function do if the `n` argument is 0?
- What should it do if the list is empty?
- What if the list is not empty and `n` is nonzero?

EXERCISE 3.4

Implement `dropWhile`, which removes elements from the List prefix as long as they match a predicate.

TIP Use pattern-matching and recursion. What should the function do if the list is empty? What if it's not empty?

EXERCISE 3.5

Not everything works out so nicely. Implement a function, `init`, that returns a `List` consisting of all but the last element of a `List`. So, given `List(1,2,3,4)`, init will return `List(1,2,3)`. Why can't this function be implemented in constant time like `tail`?

TIP Consider using simple recursion here, even though it is naive and will result in stack overflows on larger lists. We will revisit this later once we have developed better tools for dealing with such situations.

EXERCISE 3.6

Can `product`, implemented using `foldRight`, immediately halt the recursion and return `0.0` if it encounters a `0.0`? Why or why not? Consider how any short-circuiting might work if you call `foldRight` with a large list. This question has deeper implications that we will return to in chapter 5.

TIP Look at the program trace from the previous example. Based on the trace, is it possible the function supplied could choose terminate the recursion early?

EXERCISE 3.7

See what happens when you pass `Nil` and `Cons` themselves to `foldRight`, like this:

```
foldRight(
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    List.of(1, 2, 3),
    List.empty<Int>(),
    { x, y -> Cons(x, y) })
```

What do you think this says about the relationship between `foldRight` and the data constructors of `List`?

TIP The first step in the trace should be represented as:

```
Cons(1, foldRight(List.of(2, 3), z, f))
```

Now follow on with each subsequent call to `foldRight`.

EXERCISE 3.12 (HARD)

Can you write `foldLeft` in terms of `foldRight`? How about the other way around? Implementing `foldRight` via `foldLeft` is useful because it lets us implement `foldRight` tail-recursively, which means it works even for large lists without overflowing the stack.

NOTE This exercise is pushing you well beyond what you currently know, so don't be too hard on yourself if you can't figure this one out yet!

TIP It is certainly possible to do both directions. For `foldLeft` in terms of `foldRight`, you should build up, using `foldRight`, some value that you can use to achieve the effect of `foldLeft`. This won't necessarily be the `B` of the return type but could be a function of signature `(B) -> B`, also know as `Identity` in category theory.

EXERCISE 3.14 (HARD)

Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists. Try to use functions we have already defined.

TIP The `foldRight` function that we previously defined will work here.

EXERCISE 3.15

Write a function that transforms a list of integers by adding `1` to each element. This should be a pure function that returns a new `List`.

TIP Use `foldRight` without resorting to recursion.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 3.16

Write a function that turns each value in a `List<Double>` into a `String`. You can use the expression `d.toString()` to convert some `d: Double` to a `String`.

TIP	Again, use `foldRight` without resorting to recursion.

EXERCISE 3.17

Write a function `map` that generalizes modifying each element in a list while maintaining the structure of the list. Use the `foldRight` variant that uses `foldLeft` in order to prevent large lists from blowing the stack.

TIP	Once more, use `foldRight` without resorting to recursion.

EXERCISE 3.18

Write a function `filter` that removes elements from a list unless they satisfy a given predicate. Use it to remove all odd numbers from a `List<Int>`.

TIP	One more time, `foldRight` is your friend!

EXERCISE 3.19

Write a function `flatMap` that works like `map` except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
fun <A, B> flatMap(xa: List<A>, f: (A) -> List<B>): List<B> = TODO()
```

TIP	Use a combination of existing functions that we have already defined.

EXERCISE 3.23

As an example, implement `hasSubsequence` for checking whether a List contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)`, `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function however comes most naturally. We'll return to this implementation in chapter 5 and hopefully improve on it.

As an extra hint, the exercise file suggests starting by implementing the following function:

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
tailrec fun <A> startsWith(l1: List<A>, l2: List<A>): Boolean = TODO()
```

Implementing `hasSubsequence` will be much easier using `startsWith`.

TIP It's good to specify some properties about these functions up front. For example, do you expect these assertions to be true?

```
xs.append(ys).startsWith(xs) shouldBe true
xs.startsWith(Nil) shouldBe true
xs.append(ys.append(zs)).hasSubsequence(ys) shouldBe true
xs.hasSubsequence(Nil) shouldBe true
```

You will find that if the answer to any one of these is "yes", then that implies something about the answer to the rest of them.

EXERCISE 3.28

Generalize `size`, `maximum`, `depth`, and `map` for `Tree`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function. Can you draw an analogy between this `fold` function and the left and right folds for `List`?

TIP The signature for `fold` is:

```
fun <A, B> fold(ta: Tree<A>, l: (A) -> B, b: (B, B) -> B): B
```

See if you can define this function, then reimplement the functions you've already written for `Tree`.

TIP When you implement the `mapF` function, you might run into a type mismatch error in a lambda telling that the compiler found a `Branch` where it requires a `Leaf`. To fix this, you will need to include explicit typing in the lambda arguments.

CHAPTER 4. HANDLING ERROR WITHOUT EXCEPTIONS

EXERCISE 4.3

Write a generic function, `map2` that combines two `Option` values using a binary function. If either `Option` value is `None`, then the return value is too.

```
fun <A, B, C> map2(a: Option<A>, b: Option<B>, f: (A, B) -> C): Option<C> =
    TODO()
```

TIP Use the `flatMap` and possibly the `map` method.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 4.4

Write a function, `sequence` that combines a list of `Options` into one `Option` containing a list of all the `Some` values in the original list. If the original list contains `None` even once, the result of the function should be `None`; otherwise the result should be `Some` with a list of all the values.

TIP	Break the list out using matching where there will be a recursive call to `sequence` in the `Cons` case. Alternatively, use the `foldRight` method to take care of the recursion for you.

EXERCISE 4.5

Implement the `traverse` function. It's straightforward to do using `map` and `sequence`, but try for a more efficient implementation that only looks at the list once. When complete, implement `sequence` by using `traverse`.

TIP	The `traverse` function can be written with explicit recursion, or use `foldRight` to do the recursion for you. Implementing `sequence` using `traverse` may be more trivial than you think.

EXERCISE 4.6

Implement versions of `map`, `flatMap`, `orElse`, and `map2` on `Either` that operate on the `Right` value.

TIP	The `map2` function that we wrote earlier for `Option` will follow the same pattern for `Either`.

EXERCISE 4.7

Implement `sequence` and `traverse` for `Either`. These should return the first error that's encountered, if there is one.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP The signature of `traverse` and `sequence` are as follows respectively:

```
fun <E, A, B> traverse(
    xs: List<A>,
    f: (A) -> Either<E, B>
): Either<E, List<B>> = TODO()

fun <E, A> sequence(es: List<Either<E, A>>): Either<E, List<A>> =
    TODO()
```

In your implementation, you can match on the list and use explicit recursion or use `foldRight` to perform the recursion for you.

EXERCISE 4.8

In the implementation found in Listing 4.8, `map2` is only able to report one error, even if both the name and the age are invalid. What would you need to change in order to report both errors? Would you change `map2` or the signature of `mkPerson`? Or could you create a new data type that captures this requirement better than `Either` does, with some additional structure? How would `orElse`, `traverse`, and `sequence` behave differently for that data type?

TIP There are a number of variations on `Option` and `Either`. If we want to accumulate multiple errors, a simple approach is a new data type that lets us keep a list of errors in the data constructor that represents failures.

CHAPTER 5. STRICTNESS AND LAZINESS

EXERCISE 5.1

Write a function to convert a `Stream` to a `List`, which will force its evaluation and let you look at it in the REPL as well as perform assertions in the unit tests provided in the source code repository. You can convert to the singly-linked `List` type that we developed in chapter 3 of this book. You can implement this and other functions that operate on a `Stream` using extension methods.

TIP Although a simple recursive solution will work, a stack overflow could occur on larger streams. An improved solution is to do this as a tail-recursive function with a list reversal at the end.

EXERCISE 5.2

Write the function `take(n)` for returning the first `n` elements of a `Stream`, and `drop(n)` for skipping the first `n` elements of a `Stream`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP	Many `Stream` functions can start by matching on the `Stream` and considering what to do in each of the two cases. These particular functions needs to first consider whether it needs to look at the stream at all.

EXERCISE 5.4

Implement `forAll`, which checks that all elements in the `Stream` match a given predicate. Your implementation should terminate the traversal as soon as it encounters a non-matching value.

TIP	Use `foldRight` to implement this.

EXERCISE 5.6 (HARD)

Implement `headOption` using `foldRight`.

TIP	Let `None: Option<A>` be the first argument to `foldRight`. Follow the types from there.

EXERCISE 5.9

Write a function that generates an infinite stream of integers, starting from `n`, then `n + 1`, `n + 2`, and so on.

TIP	The example function `ones` is recursive, how could you define `from` recursively?

EXERCISE 5.10

Write a function `fibs` that generates the infinite stream of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, and so on.

TIP	Chapter two discussed writing loops functionally, using a recursive helper function. Consider using the same approach here.

EXERCISE 5.11

Write a more general stream-building function called `unfold`. It takes an initial state, and a function for producing both the next state and the next value in the generated stream.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP	Review the techniques you used in exercise 4.1 for working with `Option`.

EXERCISE 5.14 (HARD)

Implement `startsWith` using functions you've written. It should check if one `Stream` is a prefix of another. For instance, `Stream(1,2,3) startsWith Stream(1,2)` would be `true`.

TIP	Try to avoid using explicit recursion. Use `zipAll` and `takeWhile`.

EXERCISE 5.15

Implement `tails` using `unfold`. For a given `Stream`, `tails` returns the `Stream` of suffixes of the input sequence, starting with the original `Stream`. For example, given `Stream.of(1,2,3)`, it would return `Stream.of(Stream.of(1,2,3), Stream.of(2,3), Stream.of(3), Stream.empty())`.

TIP	Try `unfold` with `this` as the starting state. You may want to handle emitting the empty `Stream` at the end as a special case.

EXERCISE 5.16 (HARD)

Generalize `tails` to the function `scanRight`, which is like a `foldRight` that returns a stream of the intermediate results. For example:

```
>>> Stream.of(1, 2, 3).scanRight(0, { a, b -> a + b }).toList()
res1: chapter3.List<kotlin.Int> = Cons(head=6, tail=Cons(head=5, tail=Cons(head=3,
                                                    tail=Cons(head=0, tail=Nil))))
```

This example should be equivalent to the expression `List.of(1+2+3+0, 2+3+0, 3+0, 0)`. Your function should reuse intermediate results so that traversing a `Stream` with `n` elements always takes time linear in `n`. Can it be implemented using `unfold`? How, or why not? Could it be implemented using another function we've written?

TIP	The function can't be implemented using `unfold`, since `unfold` generates elements of the `Stream` from left to right. It *can* be implemented using `foldRight` though.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

CHAPTER 6. PURELY FUNCTIONAL STATE

EXERCISE 6.2

Write a function to generate a `Double` between `0` and `1`, not including `1`. In additon to the function you already developed, you can use `Int.MAX_VALUE` to obtain the maximum positive integer value, and you can use `x.toDouble()` to convert an `x: Int` to a `Double`.

TIP	Use `nonNegativeInt` to generate a random integer between `0` and `Int.MAX_VALUE`, inclusive. Then map that to the range of doubles from 0 to 1.

EXERCISE 6.5

Use `map` to reimplement `doubleR` in a more elegant way. See exercise 6.2.

TIP	This is an application of `map` over `nonNegativeInt` or `nextInt`.

EXERCISE 6.6

Write the implementation of `map2` based on the following signature. This function takes two actions, `ra` and `rb`, and a function `f` for combining their results, and returns a new action that combines them.

TIP	Start by accepting an RNG. Note that you have a choice in which RNG to pass to which function, and in what order. Think about what you expect the behavior to be, and whether your implementation meets that expectation.

EXERCISE 6.7 (HARD)

Hard: If you can combine two RNG transitions, you should be able to combine a whole list of them. Implement `sequence` for combining a `List` of transitions into a single transition. Use it to reimplement the `ints` function you wrote before. For the sake of simplicity in this exercise, it is acceptable to write `ints` with recursion to build a list with `x` repeated `n` times.

TIP	You need to recursively iterate over the list. Remember that you can use `foldLeft` or `foldRight` instead of writing a recursive definition. You can also reuse the `map2` function you just wrote. As a test case for your implementation, we should expect `sequence(List.of(unit(1), unit(2), unit(3)))(r).first` to return `List(1, 2, 3)`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 6.8

Implement `flatMap`, and then use it to implement `nonNegativeLessThan`.

TIP	The implementation using `flatMap` will be almost identical to the failed one where we tried to use `map`.

EXERCISE 6.9

Reimplement `map` and `map2` in terms of `flatMap`. The fact that this is possible is what we're referring to when we say that `flatMap` is more *powerful* than `map` and `map2`.

TIP	`mapF`: your solution will be similar to `nonNegativeLessThan`. `map2F`: your solution to `map2` for the `Option` data type should give you some ideas.

EXERCISE 6.10

Generalize the functions `unit`, `map`, `map2`, `flatMap`, and `sequence`. Add them as methods on the `State` data class where possible. Otherwise you should put them in the `State` companion object.

TIP Use the specialized functions for `Rand` as inspiration.

Recall that if you have a `f : (S) -> Pair(A,S)`, you can create a `State<S,A>` just by writing `State(f)`. The function can also be declared inline with a lambda:

```
State { s: S ->
  ...
  Pair(a,s2)
}
```

CHAPTER 7. PURELY FUNCTIONAL PARALLELISM

EXERCISE 7.1

`Par.map2` is a new higher-order function for combining the result of two parallel computations. What is its signature? Give the most general signature possible (don't assume it works only for `Int`).

TIP	The function shouldn't require that the two Par inputs have the same type.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 7.2

At any point while evolving an API, you can start thinking about possible *representations* for the abstract types that appear. Try to come up with a representation for `Par` that makes it possible to implement the functions of our API.

TIP	What if run were backed by a `java.util.concurrent.ExecutorService`? You may want to spend some time looking through the `java.util.concurrent` package to see what other useful things you can find.

EXERCISE 7.3

Fix the implementation of `map2` so that it respects the contract of timeouts on `Future`.

TIP	In order to respect timeouts, we'd need a new `Future` implementation that records the amount of time spent evaluating one future, then subtracts that time from the available time allocated for evaluating the other future.

EXERCISE 7.5

Write this function, called `sequence`. No additional primitives are required. Do not call `run`.

TIP	One possible implementation will be very similar in structure to a function we've implemented previously for `Option`.

EXERCISE 7.7 (HARD)

Take a look through the various static methods in `Executors` to get a feel for the different implementations of `ExecutorService` that exist. Then, before continuing, go back and revisit your implementation of `fork` and try to find a counterexample, or convince yourself that the law holds for your implementation.

TIP	There is a problem is with fixed size thread pools. What happens if the thread pool is bounded to be of exactly size 1?

CHAPTER 8. PROPERTY-BASED TESTING

EXERCISE 8.4

Implement `Gen.choose` using this representation of `Gen`. It should generate integers in the range `start` to `stopExclusive`. Feel free to use functions you've already written.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun choose(start: Int, stopExclusive: Int): Gen<Int> = TODO()
```

TIP Consider using the `nonNegativeInt` method from chapter 6 to implement this generator.

EXERCISE 8.5

Let's see what else we can implement using this representation of `Gen`. Try implementing `unit`, `boolean`, and `listOfN` with the following signatures, once again drawing on functions previously written:

```
fun <A> unit(a: A): Gen<A> = TODO()

fun boolean(): Gen<Boolean> = TODO()

fun <A> listOfN(n: Int, ga: Gen<A>): Gen<List<A>> = TODO()
```

TIP We can draw heavily on the `State` API that we developed in chapter 6 for this exercise. We had a method that could provide random boolean values that might come in handy for our `boolean()` generator. Could we also reuse `State.sequence()` somehow?

EXERCISE 8.6

Implement `flatMap`, and then use it to implement this more dynamic version of `listOfN`. Place `flatMap` and `listOfN` in the `Gen` data class as shown.

TIP Try using the previous implementation of `listOfN` from Exercise 8.5 in addition to `flatMap` in your solution.

EXERCISE 8.9

Now that we have a representation of `Prop`, implement `and` and `or` for composing `Prop` values. Notice that in the case of an `or` failure, we don't know which property was responsible, the left or the right. Can you devise a way of handling this?

TIP Determining which property was responsible for the failure could be achieved by allowing `Prop` values to tag or label the messages that are propagated on failure.

EXERCISE 8.12

Implement a `listOf` combinator on `Gen` that doesn't accept an explicit size and should return an `SGen` instead of a `Gen`. The implementation should generate lists of the size provided to the `SGen`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP	Consider using the `listOfN` function you wrote before.

EXERCISE 8.13

Define `nonEmptyListOf` for generating nonempty lists, and then update your specification of `max` to use this generator.

TIP	You could use `listOfN` one more time.

EXERCISE 8.16

Express the property about `fork` from chapter 7 that `fork(x) == x`.

TIP	Use the `Gen<Par<Int>>` generator from the previous exercise.

CHAPTER 9. PARSER COMBINATORS

EXERCISE 9.1

Using `product`, implement the now-familiar combinator `map2`. In turn, use this to implement `many1` in terms of `many`.

TIP	Consider mapping over the result of `product`.

EXERCISE 9.2 (HARD)

Try coming up with laws to specify the behavior of `product`.

TIP	Multiplication of numbers is always *associative*, so `(a * b) * c` is the same as `a * (b * c)`. Is this property analogous to parsers? What is there to say about the relationship between `map` and `product`?

EXERCISE 9.7

Implement `product` and `map2` in terms of `flatMap` and `map`.

TIP	Try to use `flatMap` and `succeed`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 9.9 (HARD)

At this point, you are going to take over the design process. You'll be creating a `Parser<JSON>` from scratch using the primitives we've defined. You don't need to worry about the representation of `Parser` just yet. As you go, you'll undoubtedly discover additional combinators and idioms, notice and factor out common patterns, and so on. Use the skills you've been developing throughout this book, and have fun!

TIP	For the tokens of your grammar, it is a good idea to skip any trailing whitespace to avoid having to deal with whitespace everywhere. Try introducing a combinator for this called `token`. When sequencing parsers with `product`, it is common to want to ignore one of the parsers in the sequence, consider introducing combinators for this purpose called `skipL` and `skipR`.

EXERCISE 9.10

Can you think of any other primitives that might be useful for specifying what error(s) in an `or` chain get reported?

TIP	Here are two options: we could return the most recent error in the `or` chain, or we could return whichever error occurred after getting furthest into the input string.

EXERCISE 9.12

Revise your implementation of `string` to provide a meaningful error message in the event of an error.

TIP	You may want `string` to report the immediate cause of failure (whichever character didn't match), as well as the overall string being parsed.

CHAPTER 10. MONOIDS

EXERCISE 10.2

Give a `Monoid` instance for combining `Option` values.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP	There is more than one implementation that meets the monoid laws in this instance. Consider implementing a `dual` helper function for `Monoid`, allowing for the combination for monoids in reverse order to deal with this duality.

EXERCISE 10.3

A function having the same argument and return type is sometimes called an *endofunction*.[53] Write a monoid for endofunctions.

TIP	We are limited in the number of ways we can combine values with `op` since it should compose functions of type `(A) -> A` for any choice of `A`. There is more than one possible implementation for `op`, but only one for `zero`.

EXERCISE 10.4

Use the property-based testing framework we developed in chapter 8 to implement properties for the monoid laws of associativity and identity. Use your properties to test some of the monoids we've written so far.

TIP	You will need to generate three values of type `A` for testing the law of associativity.

EXERCISE 10.5

The function `foldMap` is used to align the types of the list elements so that a `Monoid` instance may be applied to the list. Implement this function.

TIP	It is possible to `map` and then `concatenate`, although this is very inefficient. A single `foldLeft` can be used instead.

EXERCISE 10.6 (HARD)

The `foldMap` function can be implemented using either `foldLeft` or `foldRight`. But you can also write `foldLeft` and `foldRight` using `foldMap`. Give it a try for fun!

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP The type of the function that is passed to `foldRight` is `(A, B) -> B`, which can be curried to `(A) -> (B) -> B`. This is a strong hint that we should use the endofunction monoid, `(B) -> B` to implement `foldRight`. The implementation of `foldLeft` is simply the dual of this operation. Don't be too concerned about efficiency in these implementations.

EXERCISE 10.7

Implement `foldMap` based on the *balanced fold* technique. Your implementation should use the strategy of splitting the sequence in two, recursively processing each half, and then adding the answers together using the provided monoid.

TIP The sequences of lengths `0` and `1` are special cases that should be dealt with separately.

EXERCISE 10.8 (HARD/OPTIONAL)

Also implement a *parallel* version of `foldMap` called `parFoldMap` using the library we developed in chapter 7.

TIP Consider the case of a partial answer. We need to know if what we have seen so far is ordered when we've only seen *some* of the elements. For an ordered sequence, every new element seen should *not* fall within the range of elements seen already.

EXERCISE 10.9 (HARD/OPTIONAL)

Use `foldMap` as developed in exercise 10.7 to detect ascending order of a `List<Int>`. This will require some creativity when deriving the appropriate `Monoid` instance.

TIP Try creating a data type which tracks the interval of the values in a given segment, as well as whether an "unordered segment" has been found. When merging the values for two segments, think about how these two pieces of information should be updated.

EXERCISE 10.13

Implement `Foldable<ForList>` using the `Foldable<F>` interface from the previous exercise.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP	The foldable `List` already has `foldLeft` and `foldRight` implementations that may be reused.

EXERCISE 10.19

A bag is like a set, except that it's represented by a map that contains one entry per element with that element as the key, and the value under that key is the number of times the element appears in the bag. For example:

```
>>> bag(listOf("a", "rose", "is", "a", "rose"))

res0: kotlin.collections.Map<kotlin.String, kotlin.Int> = {a=2, rose=2, is=1}
```

TIP	Consider using `mapMergeMonoid` along with another monoid that was developed earlier in the chapter to achieve this binning.

CHAPTER 11. MONADS AND FUNCTORS

EXERCISE 11.1

Write monad instances for `Par`, `Option` and `List`. Additionally, provide monad instances for `arrow.core.ListK` and `arrow.core.SequenceK`.

NOTE	The `ListK` and `SequenceK` types provided by Arrow are wrapper classes that turn their platform equivalents, `List` and `Sequence`, into fully equipped type constructors.

TIP	The `unit` and `flatMap` combinators have already been implemented in various ways for these types. Simply call them from your `Monad` implementation.

EXERCISE 11.2 (HARD)

`State` looks like it could be a monad too, but it takes two type arguments, namely `S` and `A`. You need a type constructor of only one argument to implement `Monad`. Try to implement a `State` monad, see what issues you run into, and think about if or how you can solve this. We'll discuss the solution later in this chapter.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP Since `State` is a `binary` type constructor, we need to *partially apply* it with the `S` type argument much like you would do with a partially applied function. Thus, it is not just one monad, but an entire *family* of monads, one for each type `S`. Consider devising a way to capture the type `S` in a type-level scope, and providing a partially applied `State` type in that scope. This should be possible using Arrow's `Kind2` interface.

EXERCISE 11.3

The `sequence` and `traverse` combinators should be pretty familiar to you by now, and your implementations of them from previous chapters are probably all very similar. Implement them once and for all on `Monad<F>`.

TIP These implementations should be very similar to those from previous chapters, only with more general types. Consider fold operations combined with the use of `unit` and `map2` on `Monad` for your solutions.

EXERCISE 11.4

Implement `replicateM` to generate a `Kind<F, List<A>>`, with the list being of length `n`.

TIP There is more than one way of writing this function. For example, try filling a `List<Kind<F, A>>` of length `n` combined with another combinator on the `Monad` interface. Alternatively, use simple recursion to build the enclosed list.

EXERCISE 11.6 (HARD)

Here's an example of a function we haven't seen before. Implement the function `filterM`. It's a bit like `filter`, except that instead of a function from `(A) -> Boolean`, we have an `(A) -> Kind<F, Boolean>`. Replacing various ordinary functions like `filter` with the monadic equivalent often yields interesting results. Implement this function, and then think about what it means for various data types such as `Par`, `Option` and `Gen`.

TIP Start by pattern matching on the argument. If the list is empty, our only choice is to return `unit(Nil)`.

EXERCISE 11.7

Implement the following Kleisli composition function in `Monad`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A, B, C> compose(
    f: (A) -> Kind<F, B>,
    g: (B) -> Kind<F, C>
): (A) -> Kind<F, C> = TODO()
```

TIP	Follow the types to the only possible implementation.

EXERCISE 11.8 (HARD)

Implement `flatMap` in terms of an abstract definition of `compose`. By this, it seems as though we've found another minimal set of monad combinators: `compose` and `unit`.

TIP	Consider what effect it would have if we assumed `A` to be `Unit`.

EXERCISE 11.9

Using the following values, prove that the identity laws expressed in terms of `compose` are equivalent to that stated in terms of `flatMap`:

```
val f: (A) -> Kind<F, A>
val x: Kind<F, A>
val v: A
```

TIP	Substitute each occurrence of `compose` by `flatMap`, then apply value `v` of type `A` to both sides of each equation.

EXERCISE 11.10

Prove that the identity laws hold for the `Option` monad.

TIP	We should again consider both `None` and `Some` cases, and expand the left and right side of the equation for each. The monadic `unit` can be expressed as `{ a: A  Some(a) }`, or the briefer `{ Some(it) }` if you prefer it.

EXERCISE 11.13 (HARD/OPTIONAL)

Restate the monad law of associativity in terms of `flatMap` using `join`, `map` and `unit`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

TIP Consider expressing your solution using the following type declarations when reworking the laws:

```
val f: (A) -> Kind<F, A>
val g: (A) -> Kind<F, A>
val x: Kind<F, A>
val y: Kind<F, Kind<F, Kind<F, A>>>
val z: (Kind<F, Kind<F, A>>) -> Kind<F, Kind<F, A>>
```

Use identity functions where possible to arrive at a reworked solution.

EXERCISE 11.16

Implement `map`, `flatMap` and `unit` as methods on this class, and give an implementation for `Monad<Id>`.

TIP Implement `ForId`, `IdOf` and a `fix()` extension function to provide a higher-kinded type so you can express `Monad<ForId>`.

EXERCISE 11.18

Express the laws that you would expect to mutually hold for `getState`, `setState`, `unit`, and `flatMap`?

TIP What would you expect `getState` to return right after you call `setState`? And what about the other way round?

EXERCISE 11.19 (HARD)

To cement your understanding of monads, give a monad instance for the `Reader` data type and explain what it means. Also, what are its primitive operations? What is the action of `flatMap`? What meaning does it give to monadic functions like `sequence`, `join`, and `replicateM`? What meaning does it give to the monadic laws?

TIP This monad is very similar to the `State` monad, except that it is "read-only". You can "ask" from it, but not "set" the `R` value that `flatMap` carries along.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Appendix B. Exercise solutions

B.1 Before you proceed to the solutions

This section contains all the solutions to the exercises in the book. Please make the best attempt possible at doing the exercises prior to skipping to this section to get the answers. The book is written in such a way that doing the exercises are a crucial part to your learning experience. Each exercise builds on the knowledge gained by the previous one. **Please only use this section to verify your answers, or to help you if you are absolutely stuck.**

CHAPTER 2. GETTING STARTED WITH FUNCTIONAL PROGRAMMING

2.1

Write a recursive function to get the *n*th Fibonacci number (en.wikipedia.org/wiki/Fibonacci_number). The first two Fibonacci numbers are 0 and 1. The *n*th number is always the sum of the previous two—the sequence begins 0, 1, 1, 2, 3, 5, 8, 13, 21. Your definition should use a local tail-recursive function.

```
fun fib(i: Int): Int {
    tailrec fun go(cnt: Int, curr: Int, nxt: Int): Int =
        if (cnt == 0)
            curr
        else go(cnt - 1, nxt, curr + nxt)
    return go(i, 0, 1)
}
```

2.2

Implement `isSorted`, which checks whether a `List<A>` is sorted according to a given comparison function. The function is preceded by two *extension properties* that add `.head` and `.tail` to any `List` value.

```
val <T> List<T>.tail: List<T>
    get() = drop(1)

val <T> List<T>.head: T
    get() = first()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> isSorted(aa: List<A>, order: (A, A) -> Boolean): Boolean {
    tailrec fun go(x: A, xs: List<A>): Boolean =
        if (xs.isEmpty()) true
        else if (!order(x, xs.head)) false
        else go(xs.head, xs.tail)

    return aa.isEmpty() || go(aa.head, aa.tail)
}
```

2.3

Let's look at another example, currying, which converts a function `f` of two arguments into a function of one argument that partially applies `f`.

```
fun <A, B, C> curry(f: (A, B) -> C): (A) -> (B) -> C =
    { a: A -> { b: B -> f(a, b) } }
```

2.4

Implement `uncurry`, which reverses the transformation of `curry`. Note that since `->` associates to the right, `(A) -> ((B) -> C)` can be written as `(A) -> (B) -> C`.

```
fun <A, B, C> uncurry(f: (A) -> (B) -> C): (A, B) -> C =
    { a: A, b: B -> f(a)(b) }
```

2.5

Implement the higher-order function that composes two functions.

```
fun <A, B, C> compose(f: (B) -> C, g: (A) -> B): (A) -> C =
    { a: A -> f(g(a)) }
```

CHAPTER 3. FUNCTIONAL DATA STRUCTURES

EXERCISE 3.1

Implement the function `tail` for removing the first element of a `List`. Note that the function takes constant time. What are different choices you could make in your implementation if the `List` is `Nil`? We'll return to this question in the next chapter.

```
fun <A> tail(xs: List<A>): List<A> =
    when (xs) {
        is Cons -> xs.tail
        is Nil ->
            throw IllegalStateException("Nil cannot have a `tail`")
    }
```

EXERCISE 3.2

Using the same idea, implement the function `setHead` for replacing the first element of a `List` with a different value.

```
fun <A> setHead(xs: List<A>, x: A): List<A> =
    when (xs) {
        is Nil ->
            throw IllegalStateException(
                "Cannot replace `head` of a Nil list"
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
            )
        is Cons -> Cons(x, xs.tail)
    }
```

EXERCISE 3.3

Generalize `tail` to the function `drop`, which removes the first `n` elements from a list. Note that this function takes time proportional only to the number of elements being dropped—we don't need to make a copy of the entire `List`.

```
tailrec fun <A> drop(l: List<A>, n: Int): List<A> =
    if (n == 0) l
    else when (l) {
        is Cons -> drop(l.tail, n - 1)
        is Nil -> throw IllegalStateException(
            "Cannot drop more elements than in list"
        )
    }
```

EXERCISE 3.4

Implement `dropWhile`, which removes elements from the List prefix as long as they match a predicate.

```
tailrec fun <A> dropWhile(l: List<A>, f: (A) -> Boolean): List<A> =
    when (l) {
        is Cons ->
            if (f(l.head)) dropWhile(l.tail, f) else l
        is Nil -> l
    }
```

EXERCISE 3.5

Not everything works out so nicely. Implement a function, `init`, that returns a `List` consisting of all but the last element of a `List`. So, given `List(1,2,3,4)`, init will return `List(1,2,3)`. Why can't this function be implemented in constant time like `tail`?

```
fun <A> init(l: List<A>): List<A> =
    when (l) {
        is Cons ->
            if (l.tail == Nil) Nil
            else Cons(l.head, init(l.tail))
        is Nil ->
            throw IllegalStateException("Cannot init Nil list")
    }
```

EXERCISE 3.6

Can `product`, implemented using `foldRight`, immediately halt the recursion and return `0.0` if it encounters a `0.0`? Why or why not? Consider how any short-circuiting might work if you call `foldRight` with a large list. This question has deeper implications that we will return to in chapter 5.

No, this is not possible! The reason is because ***before*** *we ever call our function,* `f`*, we evaluate its argument, which in the case of* `foldRight` *means traversing the list all the way to the end.*

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

*We need **non-strict** evaluation to support early termination—we discuss this in chapter 5.*

EXERCISE 3.7

See what happens when you pass `Nil` and `Cons` themselves to `foldRight`, like this:

```
fun <A, B> foldRight(xs: List<A>, z: B, f: (A, B) -> B): B =
    when (xs) {
        is Nil -> z
        is Cons -> f(xs.head, foldRight(xs.tail, z, f))
    }

val f = { x: Int, y: List<Int> -> Cons(x, y) }
val z = Nil as List<Int>

val trace = {
    foldRight(List.of(1, 2, 3), z, f)
    Cons(1, foldRight(List.of(2, 3), z, f))
    Cons(1, Cons(2, foldRight(List.of(3), z, f)))
    Cons(1, Cons(2, Cons(3, foldRight(List.empty(), z, f))))
    Cons(1, Cons(2, Cons(3, Nil)))
}
```

What do you think this says about the relationship between `foldRight` and the data constructors of `List`?

Replacing `z` and `f` with `Nil` and `Cons` respectively when invoking `foldRight` results in `xs` being copied.

EXERCISE 3.8

Compute the length of a list using `foldRight`.

```
fun <A> length(xs: List<A>): Int =
    foldRight(xs, 0, { _, acc -> 1 + acc })
```

EXERCISE 3.9

Our implementation of `foldRight` is not tail-recursive and will result in a `StackOverflowError` for large lists (we say it's not stack-safe). Convince yourself that this is the case, and then write another general list-recursion function, `foldLeft`, that is tail-recursive, using the techniques we discussed in the previous chapter.

```
tailrec fun <A, B> foldLeft(xs: List<A>, z: B, f: (B, A) -> B): B =
    when (xs) {
        is Nil -> z
        is Cons -> foldLeft(xs.tail, f(z, xs.head), f)
    }
```

EXERCISE 3.10

Write `sum`, `product`, and a function to compute the length of a list using `foldLeft`.

```
fun sumL(xs: List<Int>): Int =
    foldLeft(xs, 0, { x, y -> x + y })
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun productL(xs: List<Double>): Double =
    foldLeft(xs, 1.0, { x, y -> x * y })

fun <A> lengthL(xs: List<A>): Int =
    foldLeft(xs, 0, { acc, _ -> acc + 1 })
```

EXERCISE 3.11

Write a function that returns the reverse of a list (given `List(1,2,3)` it returns `List(3,2,1)`). See if you can write it using a fold.

```
fun <A> reverse(xs: List<A>): List<A> =
    foldLeft(xs, List.empty(), { t: List<A>, h: A -> Cons(h, t) })
```

EXERCISE 3.12 (HARD)

Can you write `foldLeft` in terms of `foldRight`? How about the other way around? Implementing `foldRight` via `foldLeft` is useful because it lets us implement `foldRight` tail-recursively, which means it works even for large lists without overflowing the stack.

```
fun <A, B> foldLeftR(xs: List<A>, z: B, f: (B, A) -> B): B =
    foldRight(
        xs,
        { b: B -> b },
        { a, g ->
            { b ->
                g(f(b, a))
            }
        })(z)

fun <A, B> foldRightL(xs: List<A>, z: B, f: (A, B) -> B): B =
    foldLeft(xs,
        { b: B -> b },
        { g, a ->
            { b ->
                g(f(a, b))
            }
        })(z)

//expanded example
typealias Identity<B> = (B) -> B

fun <A, B> foldLeftRDemystified(
    ls: List<A>,
    acc: B,
    combiner: (B, A) -> B
): B {

    val identity: Identity<B> = { b: B -> b }

    val combinerDelayer: (A, Identity<B>) -> Identity<B> =
        { a: A, delayedExec: Identity<B> ->
            { b: B ->
                delayedExec(combiner(b, a))
            }
        }

    val chain: Identity<B> = foldRight(ls, identity, combinerDelayer)

    return chain(acc)
}
```

`foldLeft` processes items in the reverse order from `foldRight`. It's cheating to use `reverse`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

here because that's implemented in terms of `foldLeft`! Instead, wrap each operation in a simple identity function to delay evaluation until later, and stack (nest) the functions so that the order of application can be reversed. We'll alias the type of this particular identity/delay function `Identity<B>` so we aren't writing `(B) -> B` everywhere.

Next, we declare a simple value of `Identity`, `innderIdentity`, which will simply act as a passthrough of its value. This function will be the identity value for the inner `foldRight`.

For each item in the `ls` list (the `a` parameter), make a new delay function which will use the combiner function (passed in as parameter to the `foldLeftR_2` function) when it is evaluated later. Each new function becomes the input to the previous `delayExec` function.

We then pass the original list `ls`, plus the simple identity function and the new `combinerDelayer` to `foldRight` via the `go` function. This will create the functions for delayed evaluation with a combiner inside each one, but will not invoke any of those functions.

Finally, the `go` function is invoked which will cause each element to be evaluated lazily.

EXERCISE 3.13

Implement `append` in terms of either `foldLeft` or `foldRight`.

```
fun <A> append(a1: List<A>, a2: List<A>): List<A> =
    foldRight(a1, a2, { x, y -> Cons(x, y) })
```

EXERCISE 3.14 (HARD)

Write a function that concatenates a list of lists into a single list. Its runtime should be linear in the total length of all lists. Try to use functions we have already defined.

```
fun <A> concat(xxs: List<List<A>>): List<A> =
    foldRight(
        xxs,
        List.empty(),
        { xs1: List<A>, xs2: List<A> ->
            foldRight(xs1, xs2, { a, ls -> Cons(a, ls) })
        })

fun <A> concat2(xxs: List<List<A>>): List<A> =
    foldRight(
        xxs,
        List.empty(),
        { xs1, xs2 ->
            append(xs1, xs2)
        })
```

EXERCISE 3.15

Write a function that transforms a list of integers by adding `1` to each element. This should be a pure function that returns a new `List`.

```
fun increment(xs: List<Int>): List<Int> =
    foldRight(
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        xs,
        List.empty(),
        { i: Int, ls ->
            Cons(i + 1, ls)
        })
```

EXERCISE 3.16

Write a function that turns each value in a `List<Double>` into a `String`. You can use the expression `d.toString()` to convert some `d: Double` to a `String`.

```
fun doubleToString(xs: List<Double>): List<String> =
    foldRight(
        xs,
        List.empty(),
        { d, ds ->
            Cons(d.toString(), ds)
        })
```

EXERCISE 3.17

Write a function `map` that generalizes modifying each element in a list while maintaining the structure of the list. Use the `foldRight` variant that uses `foldLeft` in order to prevent large lists from blowing the stack.

```
fun <A, B> map(xs: List<A>, f: (A) -> B): List<B> =
    foldRightL(
        xs,
        List.empty(),
        { a, xa -> Cons(f(a), xa) })
```

EXERCISE 3.18

Write a function `filter` that removes elements from a list unless they satisfy a given predicate. Use it to remove all odd numbers from a `List<Int>`.

```
fun <A> filter(xs: List<A>, f: (A) -> Boolean): List<A> =
    foldRight(
        xs,
        List.empty(),
        { a, ls ->
            if (f(a)) Cons(a, ls)
            else ls
        })
```

EXERCISE 3.19

Write a function `flatMap` that works like `map` except that the function given will return a list instead of a single result, and that list should be inserted into the final resulting list. Here is its signature:

```
fun <A, B> flatMap(xa: List<A>, f: (A) -> List<B>): List<B> =
    foldRight(
        xa,
        List.empty(),
        { a, lb ->
            append(f(a), lb)
        })
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A, B> flatMap2(xa: List<A>, f: (A) -> List<B>): List<B> =
    foldRight(
        xa,
        List.empty(),
        { a, xb ->
            foldRight(f(a), xb, { b, lb -> Cons(b, lb) })
        })
```

EXERCISE 3.20

Use `flatMap` to implement `filter`.

```
fun <A> filter2(xa: List<A>, f: (A) -> Boolean): List<A> =
    flatMap(xa) { a ->
        if (f(a)) List.of(a) else List.empty()
    }
```

EXERCISE 3.21

Write a function that accepts two lists and constructs a new list by adding corresponding elements. For example, `List(1,2,3)` and `List(4,5,6)` become `List(5,7,9)`.

```
fun add(xa: List<Int>, xb: List<Int>): List<Int> =
    when (xa) {
        is Nil -> Nil
        is Cons -> when (xb) {
            is Nil -> Nil
            is Cons ->
                Cons(xa.head + xb.head, add(xa.tail, xb.tail))
        }
    }
```

EXERCISE 3.22

Generalize the function you just wrote so that it's not specific to integers or addition. Name your generalized function `zipWith`.

```
fun <A> zipWith(xa: List<A>, xb: List<A>, f: (A, A) -> A): List<A> =
    when (xa) {
        is Nil -> Nil
        is Cons -> when (xb) {
            is Nil -> Nil
            is Cons -> Cons(
                f(xa.head, xb.head),
                zipWith(xa.tail, xb.tail, f)
            )
        }
    }
```

EXERCISE 3.23

As an example, implement `hasSubsequence` for checking whether a List contains another `List` as a subsequence. For instance, `List(1,2,3,4)` would have `List(1,2)`, `List(2,3)`, and `List(4)` as subsequences, among others. You may have some difficulty finding a concise purely functional implementation that is also efficient. That's okay. Implement the function however comes most naturally. We'll return to this implementation in chapter 5 and hopefully improve on it.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
tailrec fun <A> startsWith(l1: List<A>, l2: List<A>): Boolean =
    when (l1) {
        is Nil -> l2 == Nil
        is Cons -> when (l2) {
            is Nil -> true
            is Cons ->
                if (l1.head == l2.head)
                    startsWith(l1.tail, l2.tail)
                else false
        }
    }

tailrec fun <A> hasSubsequence(xs: List<A>, sub: List<A>): Boolean =
    when (xs) {
        is Nil -> false
        is Cons ->
            if (startsWith(xs, sub))
                true
            else hasSubsequence(xs.tail, sub)
    }
```

EXERCISE 3.24

Write a function `size` that counts the number of nodes (leaves and branches) in a tree.

```
fun <A> size(tree: Tree<A>): Int =
    when (tree) {
        is Leaf -> 1
        is Branch -> 1 + size(tree.left) + size(tree.right)
    }
```

EXERCISE 3.25

Write a function `maximum` that returns the maximum element in a `Tree<Int>`.

```
fun maximum(tree: Tree<Int>): Int =
    when (tree) {
        is Leaf -> tree.value
        is Branch -> maxOf(maximum(tree.left), maximum(tree.right))
    }
```

EXERCISE 3.26

Write a function `depth` that returns the maximum path length from the root of a tree to any leaf.

```
fun depth(tree: Tree<Int>): Int =
    when (tree) {
        is Leaf -> 0
        is Branch -> 1 + maxOf(depth(tree.left), depth(tree.right))
    }
```

EXERCISE 3.27

Write a function `map`, analogous to the method of the same name on `List`, that modifies each element in a tree with a given function.

```
fun <A, B> map(tree: Tree<A>, f: (A) -> B): Tree<B> =
    when (tree) {
        is Leaf -> Leaf(f(tree.value))
        is Branch -> Branch(
            map(tree.left, f),
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
            map(tree.right, f)
        )
    }
```

EXERCISE 3.28

Generalize `size`, `maximum`, `depth`, and `map` for `Tree`, writing a new function `fold` that abstracts over their similarities. Reimplement them in terms of this more general function. Can you draw an analogy between this `fold` function and the left and right folds for `List`?

```
fun <A, B> fold(ta: Tree<A>, l: (A) -> B, b: (B, B) -> B): B =
    when (ta) {
        is Leaf -> l(ta.value)
        is Branch -> b(fold(ta.left, l, b), fold(ta.right, l, b))
    }

fun <A> sizeF(ta: Tree<A>): Int =
    fold(ta, { 1 }, { b1, b2 -> 1 + b1 + b2 })

fun maximumF(ta: Tree<Int>): Int =
    fold(ta, { a -> a }, { b1, b2 -> maxOf(b1, b2) })

fun <A> depthF(ta: Tree<A>): Int =
    fold(ta, { 0 }, { b1, b2 -> 1 + maxOf(b1, b2) })

fun <A, B> mapF(ta: Tree<A>, f: (A) -> B): Tree<B> =
    fold(ta, { a: A -> Leaf(f(a)) },
        { b1: Tree<B>, b2: Tree<B> -> Branch(b1, b2) })
```

CHAPTER 4. HANDLING ERROR WITHOUT EXCEPTIONS

EXERCISE 4.1

Implement all of the following functions on `Option`. As you implement each function, try to think about what it means and in what situations you'd use it.

```
fun <A, B> Option<A>.map(f: (A) -> B): Option<B> =
    when (this) {
        is None -> None
        is Some -> Some(f(this.get))
    }

fun <A> Option<A>.getOrElse(default: () -> A): A =
    when (this) {
        is None -> default()
        is Some -> this.get
    }

fun <A, B> Option<A>.flatMap(f: (A) -> Option<B>): Option<B> =
    this.map(f).getOrElse { None }

fun <A> Option<A>.orElse(ob: () -> Option<A>): Option<A> =
    this.map { Some(it) }.getOrElse { ob() }

fun <A> Option<A>.filter(f: (A) -> Boolean): Option<A> =
    this.flatMap { a -> if (f(a)) Some(a) else None }
```

Alternative approaches:

```
fun <A, B> Option<A>.flatMap_2(f: (A) -> Option<B>): Option<B> =
    when (this) {
        is None -> None
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        is Some -> f(this.get)
    }

fun <A> Option<A>.orElse_2(ob: () -> Option<A>): Option<A> =
    when (this) {
        is None -> ob()
        is Some -> this
    }

fun <A> Option<A>.filter_2(f: (A) -> Boolean): Option<A> =
    when (this) {
        is None -> None
        is Some ->
            if (f(this.get)) this
            else None
    }
```

EXERCISE 4.2

Implement the `variance` function in terms of `flatMap`. If the mean of a sequence is `m`, the variance is the mean of `x` minus `m` to the power of `2` for each element of `x` in the sequence. In code, this will be `(x - m).pow(2)`. The `mean` method developed in Listing 4.2 may be used to implement this.

```
//using `mean` method from listing 4.2
fun mean(xs: List<Double>): Option<Double> =
    if (xs.isEmpty()) None
    else Some(xs.sum() / xs.size())

fun variance(xs: List<Double>): Option<Double> =
    mean(xs).flatMap { m ->
        mean(xs.map { x ->
            (x - m).pow(2)
        })
    }
```

EXERCISE 4.3

Write a generic function, `map2` that combines two `Option` values using a binary function. If either `Option` value is `None`, then the return value is too.

```
fun <A, B, C> map2(
    oa: Option<A>,
    ob: Option<B>,
    f: (A, B) -> C
): Option<C> =
    oa.flatMap { a ->
        ob.map { b ->
            f(a, b)
        }
    }
```

EXERCISE 4.4

Write a function, `sequence` that combines a list of `Options` into one `Option` containing a list of all the `Some` values in the original list. If the original list contains `None` even once, the result of the function should be `None`; otherwise the result should be `Some` with a list of all the values.

```
fun <A> sequence(
    xs: List<Option<A>>
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
): Option<List<A>> =
    xs.foldRight(Some(Nil),
        { oa1: Option<A>, oa2: Option<List<A>> ->
            map2(oa1, oa2) { a1: A, a2: List<A> ->
                Cons(a1, a2)
            }
        })
```

EXERCISE 4.5

Implement the `traverse` function. It's straightforward to do using `map` and `sequence`, but try for a more efficient implementation that only looks at the list once. When complete, implement `sequence` by using `traverse`.

```
   fun <A, B> traverse(
       xa: List<A>,
       f: (A) -> Option<B>
   ): Option<List<B>> =
       when (xa) {
           is Nil -> Some(Nil)
           is Cons ->
               map2(f(xa.head), traverse(xa.tail, f)) { b, xb ->
                   Cons(b, xb)
               }
       }

   fun <A> sequence(xs: List<Option<A>>): Option<List<A>> =
       traverse(xs) { it }
```

EXERCISE 4.6

Implement versions of `map`, `flatMap`, `orElse`, and `map2` on `Either` that operate on the `Right` value.

```
fun <E, A, B> Either<E, A>.map(f: (A) -> B): Either<E, B> =
    when (this) {
        is Left -> this
        is Right -> Right(f(this.value))
    }

fun <E, A> Either<E, A>.orElse(f: () -> Either<E, A>): Either<E, A> =
    when (this) {
        is Left -> f()
        is Right -> this
    }

fun <E, A, B> Either<E, A>.flatMap(f: (A) -> Either<E, B>): Either<E, B> =
    when (this) {
        is Left -> this
        is Right -> f(this.value)
    }

fun <E, A, B, C> map2(
    ae: Either<E, A>,
    be: Either<E, B>,
    f: (A, B) -> C
): Either<E, C> =
    ae.flatMap { a -> be.map { b -> f(a, b) } }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 4.7

Implement `sequence` and `traverse` for `Either`. These should return the first error that's encountered, if there is one.

```
fun <E, A, B> traverse(
    xs: List<A>,
    f: (A) -> Either<E, B>
): Either<E, List<B>> =
    when (xs) {
        is Nil -> Right(Nil)
        is Cons ->
            map2(f(xs.head), traverse(xs.tail, f)) { b, xb ->
                Cons(b, xb)
            }
    }

fun <E, A> sequence(es: List<Either<E, A>>): Either<E, List<A>> =
    traverse(es) { it }
```

EXERCISE 4.8

In the implementation found in Listing 4.8, `map2` is only able to report one error, even if both the name and the age are invalid. What would you need to change in order to report *both* errors? Would you change `map2` or the signature of `mkPerson`? Or could you create a new data type that captures this requirement better than `Either` does, with some additional structure? How would `orElse`, `traverse`, and `sequence` behave differently for that data type?

There are a number of variations on `Option` and `Either`. If we want to accumulate multiple errors, a simple approach is a new data type that lets us keep a list of errors in the data constructor that represents failures:

```
sealed class Partial<out A, out B>

data class Failures<out A>(val get: List<A>) : Partial<A, Nothing>()
data class Success<out B>(val get: B) : Partial<Nothing, B>()
```

There is a type very similar to this called `Validated` in the Arrow library. You can implement `map`, `map2`, `sequence`, and so on for this type in such a way that errors are accumulated when possible (`flatMap` is unable to accumulate errors—can you see why?). This idea can even be generalized further—we don't need to accumulate failing values into a list; we can accumulate values using any user-supplied binary function. It's also possible to use `Either<List<E>,_>` directly to accumulate errors, using different implementations of helper functions like `map2` and `sequence`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

CHAPTER 5. STRICTNESS AND LAZINESS

EXERCISE 5.1

Write a function to convert a `Stream` to a `List`, which will force its evaluation and let you look at it in the REPL as well as perform assertions in the unit tests provided in the source code repository. You can convert to the singly-linked `List` type that we developed in chapter 3 of this book. You can implement this and other functions that operate on a `Stream` using extension methods.

```
//Unsafe! Naive solution could cause a stack overflow.
fun <A> Stream<A>.toListUnsafe(): List<A> = when (this) {
    is Empty -> NilL
    is Cons -> ConsL(this.head(), this.tail().toListUnsafe())
}

//Use tailrec in combination with reverse for a safer implementation
fun <A> Stream<A>.toList(): List<A> {
    tailrec fun go(xs: Stream<A>, acc: List<A>): List<A> = when (xs) {
        is Empty -> acc
        is Cons -> go(xs.tail(), ConsL(xs.head(), acc))
    }
    return reverse(go(this, NilL))
}
```

EXERCISE 5.2

Write the function `take(n)` for returning the first `n` elements of a `Stream`, and `drop(n)` for skipping the first `n` elements of a `Stream`.

```
   fun <A> Stream<A>.take(n: Int): Stream<A> {
       fun go(xs: Stream<A>, n: Int): Stream<A> = when (xs) {
           is Empty -> empty()
           is Cons ->
               if (n == 0) empty()
               else cons(xs.head, { go(xs.tail(), n - 1) })
       }
       return go(this, n)
   }

   fun <A> Stream<A>.drop(n: Int): Stream<A> {
       tailrec fun go(xs: Stream<A>, n: Int): Stream<A> = when (xs) {
           is Empty -> empty()
           is Cons ->
               if (n == 0) xs
               else go(xs.tail(), n - 1)
       }
       return go(this, n)
   }
```

EXERCISE 5.3

Write the function `takeWhile` for returning all starting elements of a `Stream` that match the given predicate.

```
fun <A> Stream<A>.takeWhile(p: (A) -> Boolean): Stream<A> =
    when (this) {
        is Empty -> empty()
        is Cons ->
            if (p(this.head()))
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
                cons(this.head, { this.tail().takeWhile(p) })
            else empty()
    }
```

EXERCISE 5.4

Implement `forAll`, which checks that all elements in the `Stream` match a given predicate. Your implementation should terminate the traversal as soon as it encounters a non-matching value.

```
fun <A> Stream<A>.forAll(p: (A) -> Boolean): Boolean =
    foldRight({ true }, { a, b -> p(a) && b() })
```

EXERCISE 5.5

Use `foldRight` to implement `takeWhile`.

```
fun <A> Stream<A>.takeWhile(p: (A) -> Boolean): Stream<A> =
    foldRight({ empty() },
        { h, t -> if (p(h)) cons({ h }, t) else t() })
```

EXERCISE 5.6 (HARD)

Implement `headOption` using `foldRight`.

```
fun <A> Stream<A>.headOption(): Option<A> =
    this.foldRight(
        { Option.empty() },
        { a, _ -> Some(a) }
    )
```

EXERCISE 5.7

Implement `map`, `filter`, `append`, and `flatMap` using `foldRight`. The `append` method should be non-strict in its argument.

```
   fun <A, B> Stream<A>.map(f: (A) -> B): Stream<B> =
       this.foldRight(
           { empty<B>() },
           { h, t -> cons({ f(h) }, t) })

   fun <A> Stream<A>.filter(f: (A) -> Boolean): Stream<A> =
       this.foldRight(
           { empty<A>() },
           { h, t -> if (f(h)) cons({ h }, t) else t() })

   fun <A> Stream<A>.append(sa: () -> Stream<A>): Stream<A> =
       foldRight(
           sa,
           { h, t -> cons({ h }, t) })

   fun <A, B> Stream<A>.flatMap(f: (A) -> Stream<B>): Stream<B> =
       foldRight(
           { empty<B>() },
           { h, t -> f(h).append(t) })
```

EXERCISE 5.8

Generalize `ones` slightly to the function `constant`, which returns an infinite `Stream` of a given value.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> constant(a: A): Stream<A> =
    Stream.cons({ a }, { constant(a) })
```

EXERCISE 5.9

Write a function that generates an infinite stream of integers, starting from `n`, then `n + 1`, `n + 2`, and so on.

```
fun from(n: Int): Stream<Int> =
    cons({ n }, { from(n + 1) })
```

EXERCISE 5.10

Write a function `fibs` that generates the infinite stream of Fibonacci numbers: 0, 1, 1, 2, 3, 5, 8, and so on.

```
fun fibs(): Stream<Int> {
    fun go(curr: Int, nxt: Int): Stream<Int> =
        cons({ curr }, { go(nxt, curr + nxt) })
    return go(0, 1)
}
```

EXERCISE 5.11

Write a more general stream-building function called `unfold`. It takes an initial state, and a function for producing both the next state and the next value in the generated stream.

```
fun <A, S> unfold(z: S, f: (S) -> Option<Pair<A, S>>): Stream<A> =
    f(z).map { pair ->
        cons({ pair.first },
            { unfold(pair.second, f) })
    }.getOrElse {
        empty()
    }
```

EXERCISE 5.12

Write `fibs`, `from`, `constant`, and `ones` in terms of `unfold`.

```
   fun fibs(): Stream<Int> =
       unfold(Pair(0, 1), { (curr, next) ->
           Some(Pair(curr, Pair(next, curr + next)))
       })

   fun from(n: Int): Stream<Int> =
       unfold(n, { a -> Some(Pair(a, a + 1)) })

   fun <A> constant(n: A): Stream<A> =
       unfold(n, { a -> Some(Pair(a, a)) })

   fun ones(): Stream<Int> =
       unfold(1, { Some(Pair(1, 1)) })
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 5.13

Use `unfold` to implement `map`, `take`, `takeWhile`, `zipWith` (as in chapter 3), and `zipAll`. The `zipAll` function should continue the traversal as long as either stream has more elements—it uses `Option` to indicate whether each stream has been exhausted.

```
fun <A, B> Stream<A>.map(f: (A) -> B): Stream<B> =
    unfold(this,
        { s: Stream<A> ->
            when (s) {
                is Cons -> Some(Pair(f(s.head()), s.tail()))
                else -> None
            }
        })

fun <A> Stream<A>.take(n: Int): Stream<A> =
    unfold(this,
        { s: Stream<A> ->
            when (s) {
                is Cons ->
                    if (n > 0)
                        Some(Pair(s.head(), s.tail().take(n - 1)))
                    else None
                else -> None
            }
        })

fun <A> Stream<A>.takeWhile(p: (A) -> Boolean): Stream<A> =
    unfold(this,
        { s: Stream<A> ->
            when (s) {
                is Cons ->
                    if (p(s.head()))
                        Some(Pair(s.head(), s.tail()))
                    else None
                else -> None
            }
        })

fun <A, B, C> Stream<A>.zipWith(
    that: Stream<B>,
    f: (A, B) -> C
): Stream<C> =
    unfold(Pair(this, that)) { (ths: Stream<A>, tht: Stream<B>) ->
        when (ths) {
            is Cons ->
                when (tht) {
                    is Cons ->
                        Some(
                            Pair(
                                f(ths.head(), tht.head()),
                                Pair(ths.tail(), tht.tail())
                            )
                        )
                    else -> None
                }
            else -> None
        }
    }

fun <A, B> Stream<A>.zipAll(
    that: Stream<B>
): Stream<Pair<Option<A>, Option<B>>> =
    unfold(Pair(this, that)) { (ths, tht) ->
        when (ths) {
            is Cons -> when (tht) {
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
                is Cons ->
                    Some(
                        Pair(
                            Pair(Some(ths.head()), Some(tht.head())),
                            Pair(ths.tail(), tht.tail())
                        )
                    )
                else ->
                    Some(
                        Pair(
                            Pair(Some(ths.head()), None),
                            Pair(ths.tail(), empty<B>())
                        )
                    )
            }
            else -> when (tht) {
                is Cons ->
                    Some(
                        Pair(
                            Pair(None, Some(tht.head())),
                            Pair(empty<A>(), tht.tail())
                        )
                    )
                else -> None
            }
        }
    }
```

EXERCISE 5.14 (HARD)

Implement `startsWith` using functions you've written. It should check if one `Stream` is a prefix of another. For instance, `Stream(1,2,3) startsWith Stream(1,2)` would be `true`.

```
fun <A> Stream<A>.startsWith(that: Stream<A>): Boolean =
    this.zipAll(that)
        .takeWhile { !it.second.isEmpty() }
        .forAll { it.first == it.second }
```

EXERCISE 5.15

Implement `tails` using `unfold`. For a given `Stream`, `tails` returns the `Stream` of suffixes of the input sequence, starting with the original `Stream`. For example, given `Stream.of(1,2,3)`, it would return `Stream.of(Stream.of(1,2,3), Stream.of(2,3), Stream.of(3), Stream.empty())`.

```
fun <A> Stream<A>.tails(): Stream<Stream<A>> =
    unfold(this) { s: Stream<A> ->
        when (s) {
            is Cons ->
                Some(Pair(s, s.tail()))
            else -> None
        }
    }
```

EXERCISE 5.16 (HARD)

Generalize `tails` to the function `scanRight`, which is like a `foldRight` that returns a stream of the intermediate results. For example:

```
>>> Stream.of(1, 2, 3).scanRight(0, { a, b -> a + b }).toList()
res1: chapter3.List<kotlin.Int> = Cons(head=6, tail=Cons(head=5, tail=Cons(head=3,
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
                                                    tail=Cons(head=0, tail=Nil))))
```

This example should be equivalent to the expression `List.of(1+2+3+0, 2+3+0, 3+0, 0)`. Your function should reuse intermediate results so that traversing a `Stream` with `n` elements always takes time linear in `n`. Can it be implemented using `unfold`? How, or why not? Could it be implemented using another function we've written?

```
fun <A, B> Stream<A>.scanRight(z: B, f: (A, () -> B) -> B): Stream<B> =
    foldRight({ Pair(z, Stream.of(z)) },
        { a: A, p0: () -> Pair<B, Stream<B>> ->
            val p1: Pair<B, Stream<B>> by lazy { p0() }
            val b2: B = f(a) { p1.first }
            Pair<B, Stream<B>>(b2, cons({ b2 }, { p1.second }))
        }).second
```

CHAPTER 6. PURELY FUNCTIONAL STATE

EXERCISE 6.1

Write a function that uses `RNG.nextInt` to generate a random integer between `0` and `Int.MAX_VALUE` (inclusive). Make sure to handle the corner case when `nextInt` returns `Int.MIN_VALUE`, which doesn't have a non-negative counterpart.

```
fun nonNegativeInt(rng: RNG): Pair<Int, RNG> {
    val (i1, rng2) = rng.nextInt()
    return Pair(if (i1 < 0) -(i1 + 1) else i1, rng2)
}
```

EXERCISE 6.2

Write a function to generate a `Double` between `0` and `1`, not including `1`. In additon to the function you already developed, you can use `Int.MAX_VALUE` to obtain the maximum positive integer value, and you can use `x.toDouble()` to convert an `x: Int` to a `Double`.

```
fun double(rng: RNG): Pair<Double, RNG> {
    val (i, rng2) = nonNegativeInt(rng)
    return Pair(i / (Int.MAX_VALUE.toDouble() + 1), rng2)
}
```

EXERCISE 6.3

Write functions to generate a `Pair<Int, Double>`, a `Pair<Double, Int>`, and a `Triple<Double, Double, Double>`. You should be able to reuse the functions you've already written.

```
   fun intDouble(rng: RNG): Pair<Pair<Int, Double>, RNG> {
       val (i, rng2) = rng.nextInt()
       val (d, rng3) = double(rng2)
       return Pair(Pair(i, d), rng3)
   }

   fun doubleInt(rng: RNG): Pair<Pair<Double, Int>, RNG> {
       val (id, rng2) = intDouble(rng)
       val (i, d) = id
       return Pair(Pair(d, i), rng2)
   }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun double3(rng: RNG): Pair<Triple<Double, Double, Double>, RNG> {
    val (d1, rng2) = doubleR(rng)
    val (d2, rng3) = doubleR(rng2)
    val (d3, rng4) = doubleR(rng3)
    return Pair(Triple(d1, d2, d3), rng4)
}
```

EXERCISE 6.4

Write a function to generate a list of random integers.

```
fun ints(count: Int, rng: RNG): Pair<List<Int>, RNG> =
    if (count > 0) {
        val (i, r1) = rng.nextInt()
        val (xs, r2) = ints(count - 1, r1)
        Pair(Cons(i, xs), r2)
    } else Pair(Nil, rng)
```

EXERCISE 6.5

Use `map` to reimplement `doubleR` in a more elegant way. See exercise 6.2.

```
val doubleR: Rand<Double> =
    map(::nonNegativeInt) { i ->
        i / (Int.MAX_VALUE.toDouble() + 1)
    }
```

EXERCISE 6.6

Write the implementation of `map2` based on the following signature. This function takes two actions, `ra` and `rb`, and a function `f` for combining their results, and returns a new action that combines them:

```
fun <A, B, C> map2(ra: Rand<A>, rb: Rand<B>, f: (A, B) -> C): Rand<C> =
    { r1: RNG ->
        val (a, r2) = ra(r1)
        val (b, r3) = rb(r2)
        Pair(f(a, b), r3)
    }
```

EXERCISE 6.7

If you can combine two RNG transitions, you should be able to combine a whole list of them. Implement `sequence` for combining a `List` of transitions into a single transition. Use it to reimplement the `ints` function you wrote before. For the sake of simplicity in this exercise, it is acceptable to write `ints` with recursion to build a list with `x` repeated `n` times.

```
//using a simpler recursive strategy, could blow the stack
fun <A> sequence(fs: List<Rand<A>>): Rand<List<A>> = { rng ->
    when (fs) {
        is Nil -> unit(List.empty<A>())(rng)
        is Cons -> {
            val (a, nrng) = fs.head(rng)
            val (xa, frng) = sequence(fs.tail)(nrng)
            Pair(Cons(a, xa), frng)
        }
    }
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
//a better approach using foldRight
fun <A> sequence2(fs: List<Rand<A>>): Rand<List<A>> =
    foldRight(fs, unit(List.empty()), { f, acc ->
        map2(f, acc, { h, t -> Cons(h, t) })
    })

fun ints2(count: Int, rng: RNG): Pair<List<Int>, RNG> {
    fun go(c: Int): List<Rand<Int>> =
        if (c == 0) Nil
        else Cons({ r -> Pair(1, r) }, go(c - 1))
    return sequence2(go(count))(rng)
}
```

EXERCISE 6.8

Implement `flatMap`, and then use it to implement `nonNegativeLessThan`.

```
fun <A, B> flatMap(f: Rand<A>, g: (A) -> Rand<B>): Rand<B> =
    { rng ->
        val (a, rng2) = f(rng)
        g(a)(rng2)
    }

fun nonNegativeIntLessThan(n: Int): Rand<Int> =
    flatMap(::nonNegativeInt) { i ->
        val mod = i % n
        if (i + (n - 1) - mod >= 0) unit(mod)
        else nonNegativeIntLessThan(n)
    }
```

EXERCISE 6.9

Reimplement `map` and `map2` in terms of `flatMap`. The fact that this is possible is what we're referring to when we say that `flatMap` is more *powerful* than `map` and `map2`.

```
   fun <A, B> mapF(ra: Rand<A>, f: (A) -> B): Rand<B> =
       flatMap(ra) { a -> unit(f(a)) }

   fun <A, B, C> map2F(
       ra: Rand<A>,
       rb: Rand<B>,
       f: (A, B) -> C
   ): Rand<C> =
       flatMap(ra) { a ->
           map(rb) { b ->
               f(a, b)
           }
       }
```

EXERCISE 6.10

Generalize the functions `unit`, `map`, `map2`, `flatMap`, and `sequence`. Add them as methods on the `State` data class where possible. Otherwise you should put them in the `State` companion object.

```
data class State<S, out A>(val run: (S) -> Pair<A, S>) {

    companion object {
        fun <S, A> unit(a: A): State<S, A> =
            State { s: S -> Pair(a, s) }

        fun <S, A, B, C> map2(
            ra: State<S, A>,
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
            rb: State<S, B>,
            f: (A, B) -> C
        ): State<S, C> =
            ra.flatMap { a ->
                rb.map { b ->
                    f(a, b)
                }
            }

        fun <S, A> sequence(fs: List<State<S, A>>): State<S, List<A>> =
            foldRight(fs, unit(List.empty<A>()),
                { f, acc ->
                    map2(f, acc) { h, t -> Cons(h, t) }
                }
            )
    }

    fun <B> map(f: (A) -> B): State<S, B> =
        flatMap { a -> unit<S, B>(f(a)) }

    fun <B> flatMap(f: (A) -> State<S, B>): State<S, B> =
        State { s: S ->
            val (a: A, s2: S) = this.run(s)
            f(a).run(s2)
        }
}
```

EXERCISE 6.11 (HARD/OPTIONAL)

To gain experience with the use of `State`, implement a finite state automaton that models a simple candy dispenser. The machine has two types of input: you can insert a coin, or you can turn the knob to dispense candy. It can be in one of two states: locked or unlocked. It also tracks how many candies are left and how many coins it contains.

```
sealed class Input

object Coin : Input()
object Turn : Input()

data class Machine(
    val locked: Boolean,
    val candies: Int,
    val coins: Int
)
```

The rules of the machine are as follows:

- Inserting a coin into a locked machine will cause it to unlock if there's any candy left.
- Turning the knob on an unlocked machine will cause it to dispense candy and become locked.
- Turning the knob on a locked machine or inserting a coin into an unlocked machine does nothing.
- A machine that's out of candy ignores all inputs.

The method `simulateMachine` should operate the machine based on the list of inputs and return the number of coins and candies left in the machine at the end. For example, if the input `Machine` has 10 coins and 5 candies, and a total of 4 candies are successfully bought, the output should be `(14, 1)`.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
val update: (Input) -> (Machine) -> Machine =
    { i: Input ->
        { s: Machine ->
            when (i) {
                is Coin ->
                    if (!s.locked || s.candies == 0) s
                    else Machine(false, s.candies, s.coins + 1)
                is Turn ->
                    if (s.locked || s.candies == 0) s
                    else Machine(true, s.candies - 1, s.coins)
            }
        }
    }

fun simulateMachine(
    inputs: List<Input>
): State<Machine, Tuple2<Int, Int>> =
    State.fx(Id.monad()) {
        val (x) = inputs.map(update).map(StateApi::modify)
            .stateSequential()
        val (s: Machine) = StateApi.get<Machine>()
        Tuple2(s.candies, s.coins)
    }
```

CHAPTER 7. PURELY FUNCTIONAL PARALLELISM

EXERCISE 7.1

`Par.map2` is a new higher-order function for combining the result of two parallel computations. What is its signature? Give the most general signature possible (don't assume it works only for `Int`).

```
fun <A, B, C> map2(
    sum: Par<A>,
    sum1: Par<B>,
    function: (A, B) -> C
): Par<C> = Par(function(sum.get, sum1.get))
```

EXERCISE 7.2

At any point while evolving an API, you can start thinking about possible *representations* for the abstract types that appear. Try to come up with a representation for `Par` that makes it possible to implement the functions of our API.

```
class Par<A>(val get: A) {
    companion object {

        fun <A> unit(a: A): Par<A> = Par(a)

        fun <A, B, C> map2(
            a1: Par<A>,
            a2: Par<B>,
            f: (A, B) -> C
        ): Par<C> = Par(f(a1.get, a2.get))

        fun <A> fork(f: () -> Par<A>): Par<A> = f()

        fun <A> lazyUnit(a: () -> A): Par<A> = Par(a())

        fun <A> run(a: Par<A>): A = a.get
    }
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 7.3

Fix the implementation of `map2` so that it respects the contract of timeouts on `Future`.

```
fun <A, B, C> map2(a: Par<A>, b: Par<B>, f: (A, B) -> C): Par<C> =
    { es: ExecutorService ->
        val fa = a(es)
        val fb = b(es)
        TimedMap2Future(fa, fb, f)
    }

data class TimedMap2Future<A, B, C>(
    val pa: Future<A>,
    val pb: Future<B>,
    val f: (A, B) -> C
) : Future<C> {

    override fun isDone(): Boolean = TODO()

    override fun get(): C = TODO()

    override fun get(to: Long, tu: TimeUnit): C {
        val timeoutMillis = TimeUnit.MILLISECONDS.convert(to, tu)

        val start = System.currentTimeMillis()
        val a = pa.get(to, tu)
        val duration = System.currentTimeMillis() - start

        val remainder = timeoutMillis - duration
        val b = pb.get(remainder, TimeUnit.MILLISECONDS)
        return f(a, b)
    }

    override fun cancel(b: Boolean): Boolean = TODO()

    override fun isCancelled(): Boolean = TODO()
}
```

EXERCISE 7.4

This API already enables a rich set of operations. As an example, using `lazyUnit` write a function to convert any function `(A) -> B` to one that evaluates its result asynchronously.

```
fun <A, B> asyncF(f: (A) -> B): (A) -> Par<B> =
    { a: A -> lazyUnit { f(a) } }
```

EXERCISE 7.5

Write this function, called `sequence`. No additional primitives are required. Do not call `run`.

Two implementation are provided. The first is a more naive approach that uses simple recursion to achieve its goals.

```
   val <T> List<T>.head: T
       get() = first()

   val <T> List<T>.tail: List<T>
       get() = this.drop(1)

   val Nil = listOf<Nothing>()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun <A> sequence1(ps: List<Par<A>>): Par<List<A>> =
    when (ps) {
        Nil -> unit(Nil)
        else -> map2(
            ps.head,
            sequence1(ps.tail)
        ) { a: A, b: List<A> ->
            listOf(a) + b
        }
    }
```

The second, and probably better approach also uses recursion, but employs a splitting technique in combination with `map2` to parallelize the processing.

```
fun <A> sequence(ps: List<Par<A>>): Par<List<A>> =
    when {
        ps.isEmpty() -> unit(Nil)
        ps.size == 1 -> map(ps.head) { listOf(it) }
        else -> {
            val (l, r) = ps.splitAt(ps.size / 2)
            map2(sequence(l), sequence(r)) { la, lb ->
                la + lb
            }
        }
    }
```

EXERCISE 7.6

Implement `parFilter`, which filters elements of a list in parallel.

```
fun <A> parFilter(sa: List<A>, f: (A) -> Boolean): Par<List<A>> {
    val pars: List<Par<A>> = sa.map { lazyUnit { it } }
    return map(sequence(pars)) { la: List<A> ->
        la.flatMap { a ->
            if (f(a)) listOf(a) else emptyList()
        }
    }
}
```

EXERCISE 7.7 (HARD)

Take a look through the various static methods in `Executors` to get a feel for the different implementations of `ExecutorService` that exist. Then, before continuing, go back and revisit your implementation of `fork` and try to find a counterexample, or convince yourself that the law holds for your implementation.

Keep reading. The issue is explained in the next paragraph.

EXERCISE 7.8 (HARD)

Show that any fixed-size thread pool can be made to deadlock given this implementation of `fork`
.

For a thread pool of size 2, `fork(fork(fork(x)))` will deadlock, and so on. Another, perhaps more interesting example is `fork(map2(fork(x), fork(y)))`. In this case, the outer task is submitted first and occupies a thread waiting for both `fork(x)` and `fork(y)`. The `fork(x)` and

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

`fork(y)` tasks are submitted and run in parallel, except that only one thread is available, resulting in deadlock.

EXERCISE 7.9 (HARD/OPTIONAL)

Our non-blocking representation doesn't currently handle errors at all. If at any point our computation throws an exception, the `run` implementation's `latch` never counts down and the exception is simply swallowed. Can you fix that?

We give a fully fleshed-out solution in the `Task` data type in the code for Chapter 13.

EXERCISE 7.10

Implement `choiceN`, followed by `choice` in terms of `choiceN`.

```
fun <A> choiceN(n: Par<Int>, choices: List<Par<A>>): Par<A> =
    { es: ExecutorService ->
        choices[n(es).get()].invoke(es)
    }

fun <A> choice(cond: Par<Boolean>, t: Par<A>, f: Par<A>): Par<A> =
    { es: ExecutorService ->
        choiceN(
            map(cond, { if (it) 1 else 0 }),
            listOf(f, t)
        )(es)
    }
```

EXERCISE 7.11

Implement a combinator called `choiceMap` that accepts a `Map<K, Par<V>>` as container.

```
fun <K, V> choiceMap(key: Par<K>, choices: Map<K, Par<V>>): Par<V> =
    { es: ExecutorService ->
        choices[key(es).get()]!!.invoke(es)
    }
```

EXERCISE 7.12

Implement this new primitive `chooser`, and then use it to implement `choice`, `choiceN` and `choiceMap`.

```
fun <A, B> chooser(pa: Par<A>, choices: (A) -> Par<B>): Par<B> =
    { es: ExecutorService ->
        choices(pa(es).get())(es)
    }
```

EXERCISE 7.13

Implement `join`. Can you see how to implement `flatMap` using `join`? And can you implement `join` using `flatMap`?

```
fun <A> join(a: Par<Par<A>>): Par<A> =
    { es: ExecutorService -> a(es).get()(es) }

fun <A, B> flatMapViaJoin(pa: Par<A>, f: (A) -> Par<B>): Par<B> =
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    join(map(pa, f))

fun <A> joinViaFlatMap(a: Par<Par<A>>): Par<A> =
    flatMap(a, { it })
```

CHAPTER 8. PROPERTY-BASED TESTING

EXERCISE 8.1

To get used to thinking about testing in this way, come up with properties that specify the implementation of a `sum: (List<Int>) -> Int` function. You don't have to write your properties down as executable Kotest code—an informal description is fine.

- The sum of the empty list is `0`.
- The sum of a list whose elements are all equal to `x` is just the list's length multiplied by `x`. We might express this as `sum(List(n){x}) == n * x`
- For any list `l`, `sum(l) == sum(l.reverse())` since addition is commutative.
- Given a list, `List(x,y,z,p,q)`, `sum(List(x,y,z,p,q)) == sum(List(x,y)) + sum(List(z,p,q))`, since addition is associative. More generally, we can partition a list into two subsequences whose sum is equal to the sum of the overall list.
- The sum of `1,2,3`…`n` is `n*(n+1)/2`.

EXERCISE 8.2

What properties specify a function that finds the maximum of a `List<Int>`?

- The max of a single element list is equal to that element.
- The max of a list is greater than or equal to all elements of the list.
- The max of a list is an element of that list.
- The max of the empty list is unspecified and should throw an error or return None.

EXERCISE 8.3

Assuming the following representation, use `check` to implement `and` as a method of `Prop`.

```
interface Prop {
    fun check(): Boolean
    fun and(p: Prop): Prop {
        val checked = this.check() && p.check()
        return object : Prop {
            override fun check() = checked
        }
    }
}
```

An anonymous instance of `Prop` is returned that is based on `this` *and* the property `p` that is passed in.

EXERCISE 8.4

Implement `Gen.choose` using this representation of `Gen`. It should generate integers in the range `start` to `stopExclusive`. Feel free to use functions you've already written.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
fun choose(start: Int, stopExclusive: Int): Gen<Int> =
    Gen(State { rng: RNG -> nonNegativeInt(rng) }
        .map { start + (it % (stopExclusive - start)) })
```

EXERCISE 8.5

Let's see what else we can implement using this representation of `Gen`. Try implementing `unit`, `boolean`, and `listOfN` with the following signatures, once again drawing on functions previously written:

```
       fun <A> unit(a: A): Gen<A> = Gen(State.unit(a))

       fun boolean(): Gen<Boolean> =
           Gen(State { rng -> nextBoolean(rng) })

       fun <A> listOfN(n: Int, ga: Gen<A>): Gen<List<A>> =
           Gen(State.sequence(List(n) { ga.sample }))
```

This solution draws heavily on the `State` API that was developed in chapter 6. We have hinged our solution on the `State.sequence()` function, which is able to convert a `List<State<S, A>>` into a `State<A, List<A>>`. When applying the list containing `n` and the wrapped `sample` to this state transition, we get back a new `State` which can subsequently be wrapped up again as a new `Gen`.

EXERCISE 8.6

Implement `flatMap`, and then use it to implement this more dynamic version of `listOfN`. Place `flatMap` and `listOfN` in the `Gen` data class as shown.

```
data class Gen<A>(val sample: State<RNG, A>) {

    companion object {
        fun <A> listOfN(gn: Gen<Int>, ga: Gen<A>): Gen<List<A>> =
            gn.flatMap { n -> listOfN(n, ga) }
    }

    fun <B> flatMap(f: (A) -> Gen<B>): Gen<B> =
        Gen(sample.flatMap { a -> f(a).sample })
}
```

EXERCISE 8.7

Implement `union` for combining two generators of the same type into one by pulling values from each generator with equal likelihood.

```
fun <A> union(ga: Gen<A>, gb: Gen<A>): Gen<A> =
    boolean().flatMap { if (it) ga else gb }
```

EXERCISE 8.8

Implement `weighted`, a version of `union` that accepts a weight for each `Gen` and generates values from each `Gen` with probability proportional to its weight.

```
fun <A> weighted(
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    pga: Pair<Gen<A>, Double>,
    pgb: Pair<Gen<A>, Double>
): Gen<A> {
    val (ga, p1) = pga
    val (gb, p2) = pgb
    val prob =
        p1.absoluteValue /
            (p1.absoluteValue + p2.absoluteValue)
    return Gen(State { rng: RNG -> double(rng) })
        .flatMap { d ->
            if (d < prob) ga else gb
        }
}
```

EXERCISE 8.9

Now that we have a representation of `Prop`, implement `and` and `or` for composing `Prop` values. Notice that in the case of an `or` failure, we don't know which property was responsible, the left or the right. Can you devise a way of handling this?

```
data class Prop(val run: (TestCases, RNG) -> Result) {
    fun and(other: Prop) = Prop { n, rng ->
        when (val prop = run(n, rng)) {
            is Passed -> other.run(n, rng)
            is Falsified -> prop
        }
    }

    fun or(other: Prop) = Prop { n, rng ->
        when (val prop = run(n, rng)) {
            is Falsified ->
                other.tag(prop.failure).run(n, rng)
            is Passed -> prop
        }
    }

    private fun tag(msg: String) = Prop { n, rng ->
        when (val prop = run(n, rng)) {
            is Falsified -> Falsified(
                "$msg: ${prop.failure}",
                prop.successes
            )
            is Passed -> prop
        }
    }
}
```

We have introduced a `tag` method to to add metadata about a left failure when an `or` condition is encountered and computation must continue. We mark or tag the property with the failure message if `Falsified` before proceeding to the right side of the `or` condition. This is very simple, but does the trick for now.

EXERCISE 8.10

Implement a helper function called `unsized` for converting `Gen` to `SGen`. You can add this as a method on `Gen`.

```
data class Gen<A>(val sample: State<RNG, A>) {
    fun unsized(): SGen<A> = SGen { _ -> this }
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 8.11

Not surprisingly, `SGen` at a minimum supports many of the same operations as `Gen`, and the implementations are rather mechanical. Define some convenience functions on `SGen` that simply delegate to the corresponding functions on `Gen`. Also provide a convenient way of invoking an `SGen`.

```
data class SGen<A>(val forSize: (Int) -> Gen<A>) {

    operator fun invoke(i: Int): Gen<A> = forSize(i)

    fun <B> map(f: (A) -> B): SGen<B> =
        SGen<B> { i -> forSize(i).map(f) }

    fun <B> flatMap(f: (A) -> Gen<B>): SGen<B> =
        SGen<B> { i -> forSize(i).flatMap(f) }
}
```

EXERCISE 8.12

Implement a `listOf` combinator on `Gen` that doesn't accept an explicit size and should return an `SGen` instead of a `Gen`. The implementation should generate lists of the size provided to the `SGen`.

```
fun listOf(): SGen<List<A>> =
    SGen { i -> Gen.listOfN(i, this) }
```

EXERCISE 8.13

Define `nonEmptyListOf` for generating nonempty lists, and then update your specification of `max` to use this generator.

```
   fun <A> nonEmptyListOf(ga: Gen<A>): SGen<List<A>> =
       SGen { i -> Gen.listOfN(max(1, i), ga) }

   val maxProp =
       Prop.forAll(nonEmptyListOf(smallInt)) { ns: List<Int> ->
           val mx = ns.max()
               ?: throw IllegalStateException("max on empty list")
           !ns.exists { it > mx }
       }
```

EXERCISE 8.14

Write a property called `maxProp` to verify the behavior of `List.sorted`, which you can use to sort (among other things) a `List<Int>`.

```
val maxProp = forAll(SGen.listOf(smallInt)) { ns ->
    val nss = ns.sorted()
    nss.isEmpty() or   ❶
            (nss.size == 1) or   ❷
            nss.zip(nss.prepend(Int.MIN_VALUE))
                .foldRight(true, { p, b ->
                    val (pa, pb) = p
                    b && (pa >= pb)
                }) and   ❸
            nss.containsAll(ns) and   ❹
            !nss.exists { !ns.contains(it) }   ❺
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

1. List may be empty
2. List may only have a single element
3. List must be ordered in ascending order
4. List must contains all elements of unsorted list
5. List may *not* contain any elements that are *not* in unsorted list

EXERCISE 8.15

Write a richer generator for `Par<Int>` which builds more deeply nested parallel computations than the simple variant we've provided so far.

```
val pint2: Gen<Par<Int>> =
    Gen.choose(0, 20).flatMap { n ->
        Gen.listOfN(n, Gen.choose(-100, 100)).map { ls ->
            ls.foldLeft(unit(0)) { pint, i ->
                fork {
                    map2(pint, unit(i)) { a, b ->
                        a + b
                    }
                }
            }
        }
    }
```

EXERCISE 8.16

Express the property about `fork` from chapter 7 that `fork(x) == x`.

```
forAllPar(pint) { x ->
    equal(fork { x }, x)
}
```

EXERCISE 8.17

Come up with some other properties that `takeWhile` should satisfy. Can you think of a good property expressing the relationship between `takeWhile` and `dropWhile`?

```
l.takeWhile(f) + l.dropWhile(f) == l
```

```
   val l = listOf(1, 2, 3, 4, 5)
   val f = { i: Int -> i < 3 }
   val res0 = l.takeWhile(f) + l.dropWhile(f)

   assert(res0 == l)
```

We want to enforce that `takeWhile` returns the longest prefix whose elements satisfy the predicate. There are various ways to state this, but the general idea is that the remaining list, if non-empty, should start with an element that does not satisfy the predicate.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

CHAPTER 9. PARSER COMBINATORS

EXERCISE 9.1

Using `product`, implement the now-familiar combinator `map2`. In turn, use this to implement `many1` in terms of `many`.

```
override fun <A, B, C> map2(
    pa: Parser<A>,
    pb: () -> Parser<B>,
    f: (A, B) -> C
): Parser<C> =
    (pa product pb).map { (a, b) -> f(a, b) }

override fun <A> many1(p: Parser<A>): Parser<List<A>> =
    map2(p, { p.many() }) { a, b -> listOf(a) + b }
```

EXERCISE 9.2 (HARD)

Try coming up with laws to specify the behavior of `product`.

```
(a product b) product c
a product (b product c)
```

The `product` combinator is associative, so both expressions are more or less equal. The only difference here is how the pairs are nested. The `(a product b) product c` parser returns a `Pair<Pair<A, B>, C>` while the `a product (b product c)` combinator returns a `Pair<A, Pair<B, C>>`. We can easily introduce some new functions called `unbiasL` and `unbiasR` to flatten these structures out into `Triples`.

```
fun <A, B, C> unbiasL(p: Pair<Pair<A, B>, C>): Triple<A, B, C> =
    Triple(p.first.first, p.first.second, p.second)

fun <A, B, C> unbiasR(p: Pair<A, Pair<B, C>>): Triple<A, B, C> =
    Triple(p.first, p.second.first, p.second.second)
```

This now allows us to express the law of associativity as follows:

```
((a product b) product c).map(::unbiasL) ==
    (a product (b product c)).map(::unbiasR)
```

We often write this bijection between two sides as `~=`, as demonstrated in the following expression:

```
(a product b) product c ~= a product (b product c)
```

Another interesting observation is the relationship between `map` and `product`. It is possible to `map` either before or after taking the `product` of two parsers without affecting the behavior.

```
a.map(::f) product b.map(::g) ==
    (a product b).map { (a1, b1) -> Pair(f(a1), g(b1)) }
```

For instance, if `a` and `b` were both `Parser<String>`, and `f` and `g` both computed the length of a

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

string, it doesn't matter if we map over the results of `a` and `b` to compute their respective lengths *before*, or whether we do it *after* applying the product.

EXERCISE 9.3

Before continuing, see if you can define `many` in terms of `or`, `map2`, and `succeed`.

```
fun <A> many(pa: Parser<A>): Parser<List<A>> =
    map2(pa, many(pa)) { a, la ->
        listOf(a) + la
    } or succeed(emptyList())
```

EXERCISE 9.4

Implement the `listOfN` combinator introduced earlier using `map2` and `succeed`.

```
fun <A> listOfN(n: Int, pa: Parser<A>): Parser<List<A>> =
    if (n > 0)
        map2(pa, listOfN(n - 1, pa)) { a, la ->
            listOf(a) + la
        }
    else succeed(emptyList())
```

EXERCISE 9.5

We could also deal with non-strictness using a separate combinator like we did in chapter 7. Provide an new combinator called `defer` and make the necessary changes to your existing combinators. What do you think of that approach in this instance?

```
fun <A> defer(pa: () -> Parser<A>): Parser<A> = TODO()

fun <A> many(pa: Parser<A>): Parser<List<A>> =
    map2(pa, defer { many(pa) }) { a, la ->
        listOf(a) + la
    } or succeed(emptyList())
```

This approach could work, but arguably causes more confusion than what it's worth. For this reason, we will not introduce it and keep our combinators free from lazily initialized parsers.

EXERCISE 9.6

Using `flatMap` and any other combinators, write the context-sensitive parser we couldn't express earlier. The result should be a `Parser<Int>` that returns the number of characters read. You can make use of a new primitive called `regex` to parse digits, which promotes a regular expression `String` to a `Parser<String>`.

```
val parser: Parser<Int> = regex("[0-9]+")
    .flatMap { digit: String ->
        val reps = digit.toInt()
        listOfN(reps, char('a')).map { _ -> reps }
    }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 9.7

Implement `product` and `map2` in terms of `flatMap` and `map`.

```
fun <A, B> product(
    pa: Parser<A>,
    pb: Parser<B>
): Parser<Pair<A, B>> =
    pa.flatMap { a -> pb.map { b -> Pair(a, b) } }

fun <A, B, C> map2(
    pa: Parser<A>,
    pb: Parser<B>,
    f: (A, B) -> C
): Parser<C> =
    pa.flatMap { a -> pb.map { b -> f(a, b) } }
```

EXERCISE 9.8

`map` is no longer primitive. Express it in terms of `flatMap` and/or other combinators.

```
fun <A, B> map(pa: Parser<A>, f: (A) -> B): Parser<B> =
    pa.flatMap { a -> succeed(f(a)) }
```

EXERCISE 9.9 (HARD)

At this point, you are going to take over the design process. You'll be creating a `Parser<JSON>` from scratch using the primitives we've defined. You don't need to worry about the representation of `Parser` just yet. As you go, you'll undoubtedly discover additional combinators and idioms, notice and factor out common patterns, and so on. Use the skills you've been developing throughout this book, and have fun!

NOTE This exercise is about defining the algebra consisting of primitive and combinator declarations only. No implementations should appear in the final solution.

```
abstract class Parsers<PE> {

    // primitives

    internal abstract fun string(s: String): Parser<String>

    internal abstract fun regex(r: String): Parser<String>

    internal abstract fun <A> slice(p: Parser<A>): Parser<String>

    internal abstract fun <A> succeed(a: A): Parser<A>

    internal abstract fun <A, B> flatMap(
        p1: Parser<A>,
        f: (A) -> Parser<B>
    ): Parser<B>

    internal abstract fun <A> or(
        p1: Parser<out A>,
        p2: () -> Parser<out A>
    ): Parser<A>
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    // other combinators

    internal abstract fun char(c: Char): Parser<Char>

    internal abstract fun <A> many(p: Parser<A>): Parser<List<A>>

    internal abstract fun <A> many1(p: Parser<A>): Parser<List<A>>

    internal abstract fun <A> listOfN(
        n: Int,
        p: Parser<A>
    ): Parser<List<A>>

    internal abstract fun <A, B> product(
        pa: Parser<A>,
        pb: () -> Parser<B>
    ): Parser<Pair<A, B>>

    internal abstract fun <A, B, C> map2(
        pa: Parser<A>,
        pb: () -> Parser<B>,
        f: (A, B) -> C
    ): Parser<C>

    internal abstract fun <A, B> map(pa: Parser<A>, f: (A) -> B): Parser<B>

    internal abstract fun <A> defer(pa: Parser<A>): () -> Parser<A>

    internal abstract fun <A> skipR(
        pa: Parser<A>,
        ps: Parser<String>
    ): Parser<A>

    internal abstract fun <B> skipL(
        ps: Parser<String>,
        pb: Parser<B>
    ): Parser<B>

    internal abstract fun <A> sep(
        p1: Parser<A>,
        p2: Parser<String>
    ): Parser<List<A>>

    internal abstract fun <A> surround(
        start: Parser<String>,
        stop: Parser<String>,
        p: Parser<A>
    ): Parser<A>
}

abstract class ParsersDsl<PE> : Parsers<PE>() {

    fun <A> Parser<A>.defer(): () -> Parser<A> = defer(this)

    fun <A, B> Parser<A>.map(f: (A) -> B): Parser<B> =
        this@ParsersDsl.map(this, f)

    fun <A> Parser<A>.many(): Parser<List<A>> =
        this@ParsersDsl.many(this)

    infix fun <A> Parser<out A>.or(p: Parser<out A>): Parser<A> =
        this@ParsersDsl.or(this, p.defer())

    infix fun <A, B> Parser<A>.product(p: Parser<B>): Parser<Pair<A, B>> =
        this@ParsersDsl.product(this, p.defer())

    infix fun <A> Parser<A>.sep(p: Parser<String>): Parser<List<A>> =
        this@ParsersDsl.sep(this, p)
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    infix fun <A> Parser<A>.skipR(p: Parser<String>): Parser<A> =
        this@ParsersDsl.skipR(this, p)

    infix fun <B> Parser<String>.skipL(p: Parser<B>): Parser<B> =
        this@ParsersDsl.skipL(this, p)

    infix fun <T> T.cons(la: List<T>): List<T> = listOf(this) + la
}

abstract class JSONParsers : ParsersDsl<ParseError>() {

    // {
    //   "Company name" : "Microsoft Corporation",
    //   "Ticker": "MSFT",
    //   "Active": true,
    //   "Price": 30.66,
    //   "Shares outstanding": 8.38e9,
    //   "Related companies": [ "HPQ", "IBM", "YHOO", "DELL", "GOOG" ]
    // }

    val JSON.parser: Parser<JSON>
        get() = succeed(this)

    val String.rp: Parser<String>
        get() = regex(this)

    val String.sp: Parser<String>
        get() = string(this)

    fun thru(s: String): Parser<String> =
        ".*?${Pattern.quote(s)}".rp

    val quoted: Parser<String> =
        "\"".sp skipL thru("\"").map { it.dropLast(1) }

    val doubleString: Parser<String> =
        "[-+]?([0-9]*\\.)?[0-9]+([eE][-+]?[0-9]+)?".rp

    val double: Parser<Double> = doubleString.map { it.toDouble() }

    val lit: Parser<JSON> =
        JNull.parser or
            double.map { JNumber(it) } or
            JBoolean(true).parser or
            JBoolean(false).parser or
            quoted.map { JString(it) }

    val value: Parser<JSON> = lit or obj() or array()

    val keyval: Parser<Pair<String, JSON>> =
        quoted product (":".sp skipL value)

    val whitespace: Parser<String> = """\s*""".rp

    val eof: Parser<String> = """\z""".rp

    fun array(): Parser<JArray> =
        surround("[".sp, "]".sp,
            (value sep ",".sp).map { vs -> JArray(vs) })

    fun obj(): Parser<JObject> =
        surround("{".sp, "}".sp,
            (keyval sep ",".sp).map { kvs -> JObject(kvs.toMap()) })

    fun <A> root(p: Parser<A>): Parser<A> = p skipR eof

    val jsonParser: Parser<JSON> =
        root(whitespace skipL (obj() or array()))
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 9.10

Can you think of any other primitives that might be useful for specifying what error(s) in an `or` chain get reported?

```
fun <A> furthest(pa: Parser<A>): Parser<A>
```

In the event of an error, returns that error after consuming the most number of characters.

```
fun <A> latest(pa: Parser<A>): Parser<A>
```

In the event of an error, returns the error that occurred most recently.

EXERCISE 9.11 (HARD)

Implement `string`, `regex`, `succeed`, and `slice` for this representation of `Parser`. Some private helper function stubs have been included to lead you in the right direction.

```
override fun string(s: String): Parser<String> =
    { state: State ->
        when (val idx =
            firstNonMatchingIndex(state.input, s, state.offset)) {
            is None ->
                Success(s, s.length)
            is Some ->
                Failure(
                    state.advanceBy(idx.t).toError("'$s'"),
                    idx.t != 0
                )
        }
    }

private fun firstNonMatchingIndex(
    s1: String,
    s2: String,
    offset: Int
): Option<Int> {
    var i = 0
    while (i < s1.length && i < s2.length) {
        if (s1[i + offset] != s2[i])
            return Some(i)
        else i += 1
    }
    return if (s1.length - offset >= s2.length) None
    else Some(s1.length - offset)
}

private fun State.advanceBy(i: Int) =
    this.copy(offset = this.offset + i)

override fun regex(r: String): Parser<String> =
    { state: State ->
        when (val prefix = state.input.findPrefixOf(r.toRegex())) {
            is Some ->
                Success(prefix.t.value, prefix.t.value.length)
            is None ->
                Failure(state.toError("regex ${r.toRegex()}"))
        }
    }

private fun String.findPrefixOf(r: Regex): Option<MatchResult> =
    r.find(this).toOption().filter { it.range.first == 0 }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    override fun <A> succeed(a: A): Parser<A> = { Success(a, 0) }

    override fun <A> slice(p: Parser<A>): Parser<String> =
        { state: State ->
            when (val result = p(state)) {
                is Success ->
                    Success(state.slice(result.consumed), result.consumed)
                is Failure -> result
            }
        }

    private fun State.slice(n: Int) =
        this.input.substring(this.offset..this.offset + n)
```

EXERCISE 9.12

Revise your implementation of `string` to provide a meaningful error message in the event of an error.

```
override fun string(s: String): Parser<String> =
    { state: State ->
        when (val idx =
            firstNonMatchingIndex(state.input, s, state.offset)) {
            is None ->
                Success(s, s.length)
            is Some ->
                Failure(
                    state.advanceBy(idx.t).toError("'$s'"),
                    idx.t != 0
                )
        }
    }
```

EXERCISE 9.13

Implement `run`, as well as any of the remaining primitives not yet implemented using our current representation of `Parser`. Try running your JSON parser on various inputs.

```
override fun <A> run(p: Parser<A>, input: String): Result<A> =
    p(Location(input))
```

EXERCISE 9.14

Come up with a nice way of formatting a `ParseError` for human consumption. There are a lot of choices to make, but a key insight is that we typically want to combine or group tags attached to the same location when presenting the error as a `String` for display.

```
data class ParseError(val stack: List<Pair<Location, String>> = emptyList()) {

    fun push(loc: Location, msg: String): ParseError =
        this.copy(stack = listOf(Pair(loc, msg)) + stack)

    fun label(s: String): ParseError =
        ParseError(latestLoc()
            .map { Pair(it, s) }
            .toList())

    private fun latest(): Option<Pair<Location, String>> = stack.lastOrNone()

    private fun latestLoc(): Option<Location> = latest().map { it.first }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    /**
     * Display collapsed error stack - any adjacent stack elements with the
     * same location are combined on one line. For the bottommost error, we
     * display the full line, with a caret pointing to the column of the
     * error.
     * Example:
     * 1.1 file 'companies.json'; array
     * 5.1 object
     * 5.2 key-value
     * 5.10 ':'
     * { "MSFT" ; 24,
     *          ^
     */
    override fun toString(): String =
        if (stack.isEmpty()) "no error message"
        else {
            val collapsed = collapseStack(stack)
            val context =
                collapsed.lastOrNone()
                    .map { "\n\n" + it.first.line }
                    .getOrElse { "" } +
                    collapsed.lastOrNone()
                        .map { "\n" + it.first.col }
                        .getOrElse { "" }

            collapsed.joinToString { (loc, msg) ->
                "${loc.line}.${loc.col} $msg"
            } + context
        }

    /* Builds a collapsed version of the given error stack -
     * messages at the same location have their messages merged,
     * separated by semicolons.
     */
    private fun collapseStack(
        stk: List<Pair<Location, String>>
    ): List<Pair<Location, String>> =
        stk.groupBy { it.first }
            .mapValues { it.value.joinToString() }
            .toList()
            .sortedBy { it.first.offset }
}
```

CHAPTER 10. MONOIDS

EXERCISE 10.1

Give `Monoid` instances for integer addition and multiplication, as well as for Boolean operators.

```
val intAddition: Monoid<Int> = object : Monoid<Int> {

    override fun combine(a1: Int, a2: Int): Int = a1 + a2

    override val nil: Int = 0
}

val intMultiplication: Monoid<Int> = object : Monoid<Int> {

    override fun combine(a1: Int, a2: Int): Int = a1 * a2

    override val nil: Int = 1
}

val booleanOr: Monoid<Boolean> = object : Monoid<Boolean> {

    override fun combine(a1: Boolean, a2: Boolean): Boolean = a1 || a2
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    override val nil: Boolean = false
}

val booleanAnd: Monoid<Boolean> = object : Monoid<Boolean> {

    override fun combine(a1: Boolean, a2: Boolean): Boolean = a1 && a2

    override val nil: Boolean = true
}
```

EXERCISE 10.2

Give a `Monoid` instance for combining `Option` values.

```
fun <A> optionMonoid(): Monoid<Option<A>> = object : Monoid<Option<A>> {

    override fun combine(a1: Option<A>, a2: Option<A>): Option<A> =
        a1.orElse { a2 }

    override val nil: Option<A> = None
}

fun <A> dual(m: Monoid<A>): Monoid<A> = object : Monoid<A> {

    override fun combine(a1: A, a2: A): A = m.combine(a2, a1)

    override val nil: A = m.nil
}

fun <A> firstOptionMonoid() = optionMonoid<A>()

fun <A> lastOptionMonoid() = dual(firstOptionMonoid<A>())
```

Notice that we have a choice in how we implement `op`. We can compose the options in either order. Both of the implementations satisfy the monoid laws, but they are not equivalent. This is true in general–that is, every monoid has a dual where the `op` combines things in the opposite order. Monoids like `booleanOr` and `intAddition` are equivalent to their duals because their `op` is commutative as well as associative.

EXERCISE 10.3

A function having the same argument and return type is sometimes called an *endofunction*.[54] Write a monoid for endofunctions.

```
fun <A> endoMonoid(): Monoid<(A) -> A> =
    object : Monoid<(A) -> A> {
        override fun combine(a1: (A) -> A, a2: (A) -> A): (A) -> A =
            { a -> a1(a2(a)) }

        override val nil: (A) -> A
            get() = { a -> a }
    }

fun <A> endoMonoidComposed(): Monoid<(A) -> A> =
    object : Monoid<(A) -> A> {
        override fun combine(a1: (A) -> A, a2: (A) -> A): (A) -> A =
            a1 compose a2

        override val nil: (A) -> A
            get() = { it }
    }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 10.4

Use the property-based testing framework we developed in chapter 8 to implement properties for the monoid laws of associativity and identity. Use your properties to test some of the monoids we've written so far.

```
fun <A> monoidLaws(m: Monoid<A>, gen: Gen<A>) =
    forAll(
        gen.flatMap { a ->
            gen.flatMap { b ->
                gen.map { c ->
                    Triple(a, b, c)
                }
            }
        }
    ) { (a, b, c) ->
        m.combine(a, m.combine(b, c)) == m.combine(m.combine(a, b), c) &&
            m.combine(m.nil, a) == m.combine(a, m.nil) &&
                m.combine(m.nil, a) == a
    }

class Exercise4 : WordSpec({
    val max = 100
    val count = 100
    val rng = SimpleRNG(42)
    val intGen = Gen.choose(-10000, 10000)

    "law of associativity" should {
        "be upheld using existing monoids" {
            monoidLaws(intAdditionMonoid, intGen)
                .check(max, count, rng) shouldBe Passed

            monoidLaws(intMultiplicationMonoid, intGen)
                .check(max, count, rng) shouldBe Passed
        }
    }
})
```

EXERCISE 10.5

The function `foldMap` is used to align the types of the list elements so that a `Monoid` instance may be applied to the list. Implement this function.

```
fun <A, B> foldMap(la: List<A>, m: Monoid<B>, f: (A) -> B): B =
    la.foldLeft(m.nil, { b, a -> m.combine(b, f(a)) })
```

EXERCISE 10.6 (HARD)

The `foldMap` function can be implemented using either `foldLeft` or `foldRight`. But you can also write `foldLeft` and `foldRight` using `foldMap`. Give it a try for fun!

```
fun <A, B> foldRight(la: Sequence<A>, z: B, f: (A, B) -> B): B =
    foldMap(la, endoMonoid()) { a: A -> { b: B -> f(a, b) } }(z)
```

The function type `(A, B)  B`, when curried is `(A)  (B)  B`. And of course `(B)  B` is a monoid for any `B` (via function composition).

```
fun <A, B> foldLeft(la: Sequence<A>, z: B, f: (B, A) -> B): B =
    foldMap(la, dual(endoMonoid())) { a: A -> { b: B -> f(b, a) } }(z)
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Folding to the left is the same except we flip the arguments to the function `f` to put the `B` on the correct side. Then we have to also "flip" the monoid so that it operates from left to right.

EXERCISE 10.7

Implement `foldMap` based on the *balanced fold* technique. Your implementation should use the strategy of splitting the sequence in two, recursively processing each half, and then adding the answers together using the provided monoid.

```
fun <A, B> foldMap(la: List<A>, m: Monoid<B>, f: (A) -> B): B =
    when {
        la.size >= 2 -> {
            val (la1, la2) = la.splitAt(la.size / 2)
            m.combine(foldMap(la1, m, f), foldMap(la2, m, f))
        }
        la.size == 1 ->
            f(la.first())
        else -> m.nil
    }
```

EXERCISE 10.8 (HARD/OPTIONAL)

Also implement a *parallel* version of `foldMap` called `parFoldMap` using the library we developed in chapter 7.

```
fun <A> par(m: Monoid<A>): Monoid<Par<A>> = object : Monoid<Par<A>> {

    override fun combine(pa1: Par<A>, pa2: Par<A>): Par<A> =
        map2(pa1, pa2) { a1: A, a2: A ->   ❶
            m.combine(a1, a2)
        }

    override val nil: Par<A>
        get() = unit(m.nil)   ❷
}

fun <A, B> parFoldMap(
    la: List<A>,
    pm: Monoid<Par<B>>,
    f: (A) -> B
): Par<B> =
    when {
        la.size >= 2 -> {
            val (la1, la2) = la.splitAt(la.size / 2)
            pm.combine(parFoldMap(la1, pm, f), parFoldMap(la2, pm, f))
        }
        la.size == 1 ->
            unit(f(la.first()))
        else -> pm.nil
    }

                parFoldMap(
                    listOf("lorem", "ipsum", "dolor", "sit"),
                    par(stringMonoid),   ❸
                    { it.toUpperCase() }
                )(es).invoke { cb -> result.set(cb) }   ❹
```

❶ Use `map2` from chapter 7 to combine two `Par` instances

❷ Use `unit` from chapter 7 to wrap `zero` in `Par`

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

❸ Promote a `Monoid<A>` to `Monoid<Par<A>>` using `par`

❹ Apply the executor service and invoke a callback function on `Future`

EXERCISE 10.9 (HARD/OPTIONAL)

Use `foldMap` as developed in exercise 10.7 to detect ascending order of a `List<Int>`. This will require some creativity when deriving the appropriate `Monoid` instance.

```
typealias TrackingState = Triple<Int, Int, Boolean>

val m = object : Monoid<Option<TrackingState>> {
    override fun combine(
        a1: Option<TrackingState>,
        a2: Option<TrackingState>
    ): Option<TrackingState> =
        when (a1) {
            is None -> a2
            is Some ->
                when (a2) {
                    is None -> a1
                    is Some -> Some(
                        Triple(
                            min(a1.t.first, a2.t.first),
                            max(a1.t.second, a2.t.second),
                            a1.t.third &&
                                a2.t.third &&
                                a1.t.second <= a2.t.first
                        )
                    )
                }
        }

    override val nil: Option<TrackingState> = None
}

fun ordered(ints: Sequence<Int>): Boolean =
    foldMap(ints, m) { i: Int -> Some(TrackingState(i, i, true)) }
        .map { it.third }
        .getOrElse { true }
```

EXERCISE 10.10

Write a monoid instance for `WC` and ensure that it meets both monoid laws.

```
val wcMonoid: Monoid<WC> = object : Monoid<WC> {
    override fun combine(a1: WC, a2: WC): WC =
        when (a1) {
            is Stub -> when (a2) {
                is Stub ->
                    Stub(a1.chars + a2.chars)
                is Part ->
                    Part(a1.chars + a2.ls, a2.words, a2.rs)
            }
            is Part -> when (a2) {
                is Stub ->
                    Part(a1.ls, a1.words, a1.rs + a2.chars)
                is Part ->
                    Part(
                        a1.ls,
                        a1.words + a2.words +
                            (if ((a1.rs + a2.ls).isEmpty()) 0 else 1),
                        a2.rs
                    )
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
            }
        }

    override val nil: WC
        get() = Stub("")
}
```

EXERCISE 10.11

Use the `WC` monoid to implement a function that counts words in a `String` by recursively splitting it into substrings and counting the words in those substrings.

```
fun wordCount(s: String): Int {

    fun wc(c: Char): WC =
        if (c.isWhitespace()) Part("", 0, "")
        else Stub("$c")

    fun unstub(s: String): Int = min(s.length, 1)

    return when (val wc = foldMap(s.asSequence(), wcMonoid) { wc(it) }) {
        is Stub -> unstub(wc.chars)
        is Part -> unstub(wc.rs) + wc.words + unstub(wc.rs)
    }
}
```

EXERCISE 10.12

Implement `foldLeft`, `foldRight` and `foldMap` on the `Foldable<F>` interface in terms of each other.

NOTE Using all these functions in terms of each other could result in undesired effects like circular references. We will remedy this in exercise 10.13.

```
interface Foldable<F> {

    fun <A, B> foldRight(fa: Kind<F, A>, z: B, f: (A, B) -> B): B =
        foldMap(fa, endoMonoid()) { a: A -> { b: B -> f(a, b) } }(z)

    fun <A, B> foldLeft(fa: Kind<F, A>, z: B, f: (B, A) -> B): B =
        foldMap(fa, dual(endoMonoid())) { a: A -> { b: B -> f(b, a) } }(z)

    fun <A, B> foldMap(fa: Kind<F, A>, m: Monoid<B>, f: (A) -> B): B =
        foldRight(fa, m.nil, { a, b -> m.combine(f(a), b) })
}
```

EXERCISE 10.13

Implement `Foldable<ForList>` using the `Foldable<F>` interface from the previous exercise.

NOTE A foldable version of the `List` implementation we developed in chapter 3 has been provided in the chapter 10 exercise boilerplate code.

```
object ListFoldable : Foldable<ForList> {

    override fun <A, B> foldRight(
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        fa: ListOf<A>,
        z: B,
        f: (A, B) -> B
    ): B =
        fa.fix().foldRight(z, f)

    override fun <A, B> foldLeft(
        fa: ListOf<A>,
        z: B,
        f: (B, A) -> B
    ): B =
        fa.fix().foldLeft(z, f)
}
```

EXERCISE 10.14

Recall that we implemented a binary `Tree` in chapter 3. Next, implement `Foldable<ForTree>`. You only need to override `foldMap` of `Foldable` to make this work, letting the provided `foldLeft` and `foldRight` methods use your new implementation.

NOTE A foldable version of `Tree` has been provided in the chapter 10 exercise boilerplate code.

```
object TreeFoldable : Foldable<ForTree> {
    override fun <A, B> foldMap(
        fa: TreeOf<A>,
        m: Monoid<B>,
        f: (A) -> B
    ): B =
        when (val t = fa.fix()) {
            is Leaf ->
                f(t.value)
            is Branch ->
                m.combine(foldMap(t.left, m, f), foldMap(t.right, m, f))
        }
}
```

EXERCISE 10.15

Write and instance of `Foldable<ForOption>`.

```
object OptionFoldable : Foldable<ForOption> {
    override fun <A, B> foldMap(
        fa: OptionOf<A>,
        m: Monoid<B>,
        f: (A) -> B
    ): B =
        when (val o = fa.fix()) {
            is None -> m.nil
            is Some -> f(o.get)
        }
}
```

EXERCISE 10.16

Any `Foldable` structure can be turned into a `List`. Write this convenience method for `Foldable<F>` using an existing method on the interface.

```
fun <A> toList(fa: Kind<F, A>): List<A> =
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    foldLeft(fa, List.empty(), { la, a -> Cons(a, la) })
```

EXERCISE 10.17

Implement the `productMonoid` by composing two monoids. Your implementation of `op` should be associative as long as `A.op` and `B.op` are both associative.

```
fun <A, B> productMonoid(
    ma: Monoid<A>,
    mb: Monoid<B>
): Monoid<Pair<A, B>> =
    object : Monoid<Pair<A, B>> {
        override fun combine(a1: Pair<A, B>, a2: Pair<A, B>): Pair<A, B> =
            Pair(
                ma.combine(a1.first, a2.first),
                mb.combine(a1.second, a2.second)
            )

        override val nil: Pair<A, B>
            get() = Pair(ma.nil, mb.nil)
    }
```

EXERCISE 10.18

Write a monoid instance for functions whose results themselves are monoids.

```
fun <A, B> functionMonoid(b: Monoid<B>): Monoid<(A) -> B> =
    object : Monoid<(A) -> B> {
        override fun combine(f: (A) -> B, g: (A) -> B): (A) -> B =
            { a: A -> b.combine(f(a), g(a)) }

        override val nil: (A) -> B =
            { a -> b.nil }
    }
```

EXERCISE 10.19

A bag is like a set, except that it's represented by a map that contains one entry per element with that element as the key, and the value under that key is the number of times the element appears in the bag. For example:

```
>>> bag(listOf("a", "rose", "is", "a", "rose"))

res0: kotlin.collections.Map<kotlin.String, kotlin.Int> = {a=2, rose=2, is=1}
```

Use monoids to compute such a bag from a `List<A>`.

```
object ListFoldable : Foldable<ForList> {

    override fun <A, B> foldRight(
        fa: ListOf<A>,
        z: B,
        f: (A, B) -> B
    ): B =
        fa.fix().foldRight(z, f)

    fun <A> bag(la: List<A>): Map<A, Int> =
        foldMap(la, mapMergeMonoid<A, Int>(intAdditionMonoid)) { a: A ->
            mapOf(a to 1)
        }
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

CHAPTER 11. MONADS AND FUNCTORS

EXERCISE 11.1

Write monad instances for `Par`, `Option` and `List`. Additionally, provide monad instances for `arrow.core.ListK` and `arrow.core.SequenceK`.

NOTE The `ListK` and `SequenceK` types provided by Arrow are wrapper classes that turn their platform equivalents, `List` and `Sequence`, into fully equipped type constructors.

```
object Monads {

    val parMonad = object : Monad<ForPar> {

        override fun <A> unit(a: A): ParOf<A> = Par.unit(a)

        override fun <A, B> flatMap(
            fa: ParOf<A>,
            f: (A) -> ParOf<B>
        ): ParOf<B> =
            fa.fix().flatMap { f(it).fix() }
    }

    val optionMonad = object : Monad<ForOption> {

        override fun <A> unit(a: A): OptionOf<A> = Some(a)

        override fun <A, B> flatMap(
            fa: OptionOf<A>,
            f: (A) -> OptionOf<B>
        ): OptionOf<B> =
            fa.fix().flatMap { f(it).fix() }
    }

    val listMonad = object : Monad<ForList> {

        override fun <A> unit(a: A): ListOf<A> = List.of(a)

        override fun <A, B> flatMap(
            fa: ListOf<A>,
            f: (A) -> ListOf<B>
        ): ListOf<B> =
            fa.fix().flatMap { f(it).fix() }
    }

    val listKMonad = object : Monad<ForListK> {

        override fun <A> unit(a: A): ListKOf<A> = ListK.just(a)

        override fun <A, B> flatMap(
            fa: ListKOf<A>,
            f: (A) -> ListKOf<B>
        ): ListKOf<B> =
            fa.fix().flatMap(f)
    }

    val sequenceKMonad = object : Monad<ForSequenceK> {

        override fun <A> unit(a: A): Kind<ForSequenceK, A> =
            SequenceK.just(a)
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        override fun <A, B> flatMap(
            fa: Kind<ForSequenceK, A>,
            f: (A) -> Kind<ForSequenceK, B>
        ): Kind<ForSequenceK, B> {
            return fa.fix().flatMap(f)
        }
    }
}
```

EXERCISE 11.2 (HARD)

`State` looks like it could be a monad too, but it takes two type arguments, namely `S` and `A`. You need a type constructor of only one argument to implement `Monad`. Try to implement a `State` monad, see what issues you run into, and think about if or how you can solve this. We'll discuss the solution later in this chapter.

```
data class State<S, out A>(val run: (S) -> Pair<A, S>) : StateOf<S, A>

sealed class ForState private constructor() {
    companion object
}

typealias StateOf<S, A> = Kind2<ForState, S, A>

fun <S, A> StateOf<S, A>.fix() = this as State<S, A>

typealias StatePartialOf<S> = Kind<ForState, S>

interface StateMonad<S> : Monad<StatePartialOf<S>> {

    override fun <A> unit(a: A): StateOf<S, A>

    override fun <A, B> flatMap(
        fa: StateOf<S, A>,
        f: (A) -> StateOf<S, B>
    ): StateOf<S, B>
}
```

EXERCISE 11.3

The `sequence` and `traverse` combinators should be pretty familiar to you by now, and your implementations of them from previous chapters are probably all very similar. Implement them once and for all on `Monad<F>`.

```
   fun <A> sequence(lfa: List<Kind<F, A>>): Kind<F, List<A>> =
       lfa.foldRight(
           unit(List.empty<A>()),
           { fa: Kind<F, A>, fla: Kind<F, List<A>> ->
               map2(fa, fla) { a: A, la: List<A> -> Cons(a, la) }
           }
       )

   fun <A, B> traverse(
       la: List<A>,
       f: (A) -> Kind<F, B>
   ): Kind<F, List<B>> =
       la.foldRight(
           unit(List.empty<B>()),
           { a: A, acc: Kind<F, List<B>> ->
               map2(f(a), acc) { b: B, lb: List<B> -> Cons(b, lb) }
           }
       )
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 11.4

Implement `replicateM` to generate a `Kind<F, List<A>>`, with the list being of length `n`.

```
fun <A> replicateM(n: Int, ma: Kind<F, A>): Kind<F, List<A>> =
    when (n) {
        0 -> unit(List.empty())
        else ->
            map2(ma, replicateM(n - 1, ma)) { m: A, ml: List<A> ->
                Cons(m, ml)
            }
    }

fun <A> _replicateM(n: Int, ma: Kind<F, A>): Kind<F, List<A>> =
    sequence(List.fill(n, ma))
```

EXERCISE 11.5

Think about how `replicateM` will behave for various choices of `F`. For example, how does it behave in the `List` monad? And what about `Option`? Describe in your own words the general meaning of `replicateM`.

For List, the `replicateM` function will generate a list of lists. It will contain all the lists of length `n` with elements selected from the input list.

For `Option`, it will generate either `Some` or `None` based on whether the input is `Some` or `None`. The `Some` case will contain a list of length `n` that repeats the element in the input `Option`.

It repeats the `ma` monadic value `n` times, and gathers the results in a single value where the monad `F` determines how values are actually combined.

EXERCISE 11.6 (HARD)

Here's an example of a function we haven't seen before. Implement the function `filterM`. It's a bit like `filter`, except that instead of a function from `(A) -> Boolean`, we have an `(A) -> Kind<F, Boolean>`. Replacing various ordinary functions like `filter` with the monadic equivalent often yields interesting results. Implement this function, and then think about what it means for various data types such as `Par`, `Option` and `Gen`.

```
fun <A> filterM(
    ms: List<A>,
    f: (A) -> Kind<F, Boolean>
): Kind<F, List<A>> =
    when (ms) {
        is Nil -> unit(Nil)
        is Cons ->
            flatMap(f(ms.head)) { succeed ->
                if (succeed) map(filterM(ms.tail, f)) { tail ->
                    Cons(ms.head, tail)
                } else filterM(ms.tail, f)
            }
    }
```

- For `Par`, `filterM` filters a list while applying the functions in parallel.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

- For `Option`, it filters a list, but allows the filtering function to fail and abort the filter computation.
- For `Gen`, it produces a generator for subsets of the input list, where the function `f` picks a "weight" for each element in the form of a `Gen<Boolean>`.

EXERCISE 11.7

Implement the following Kleisli composition function in `Monad`.

```
fun <A, B, C> compose(
    f: (A) -> Kind<F, B>,
    g: (B) -> Kind<F, C>
): (A) -> Kind<F, C> =
    { a: A -> flatMap(f(a)) { b: B -> g(b) } }
```

EXERCISE 11.8 (HARD)

Implement `flatMap` in terms of an abstract definition of `compose`. By this, it seems as though we've found another minimal set of monad combinators: `compose` and `unit`.

```
fun <A, B> flatMap(fa: Kind<F, A>, f: (A) -> Kind<F, B>): Kind<F, B> =
    compose<Unit, A, B>({ _ -> fa }, f)(Unit)
```

EXERCISE 11.9

Using the following values, prove that the left and right identity laws expressed in terms of `compose` are equivalent to that stated in terms of `flatMap`:

```
val f: (A) -> Kind<F, A>
val x: Kind<F, A>
val v: A
```

The right identity law can be reduced as follows:

```
compose(f, { a: A -> unit(a) })(v) == f(v)
{ b: A -> flatMap(f(b), { a: A -> unit(a) }) }(v) == f(v)
flatMap(f(v)) { a: A -> unit(a) } == f(v)
flatMap(x) { a: A -> unit(a) } == x
```

The left identity law can be reduced as follows:

```
compose({ a: A -> unit(a) }, f)(v) == f(v)
{ b: A -> flatMap({ a: A -> unit(a) }(b), f) }(v) == f(v)
{ b: A -> flatMap(unit(b), f) }(v) == f(v)
flatMap(unit(v), f) == f(v)
```

The final proofs can therefore be expressed as:

```
flatMap(x) { a -> unit(a) } == x
flatMap(unit(v), f) == f(v)
```

EXERCISE 11.10

Prove that the identity laws hold for the `Option` monad.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
        flatMap(None) { a: A -> Some(a) } == None
        None == None

        flatMap(Some(v)) { a: A -> Some(a) } == Some(v)
        Some(v) == Some(v)

        flatMap(Some(None)) { a -> Some(a) } == Some(None)
        Some(None) == Some(None)

        flatMap(Some(Some(v))) { a -> Some(a) } == Some(Some(v))
        Some(Some(v)) == Some(Some(v))
```

EXERCISE 11.11

Monadic combinators can be expressed in another minimal set, namely `map`, `unit`, and `join`. Implement the `join` combinator in terms of `flatMap`.

```
fun <A> join(mma: Kind<F, Kind<F, A>>): Kind<F, A> =
    flatMap(mma) { ma -> ma }
```

EXERCISE 11.12

Either `flatMap` or `compose` may now be implemented in terms of `join`. For the sake of this exercise, implement both.

```
   fun <A, B> flatMap(fa: Kind<F, A>, f: (A) -> Kind<F, B>): Kind<F, B> =
       join(map(fa, f))

   fun <A, B, C> compose(
       f: (A) -> Kind<F, B>,
       g: (B) -> Kind<F, C>
   ): (A) -> Kind<F, C> =
       { a -> join(map(f(a), g)) }
```

EXERCISE 11.13 (HARD/OPTIONAL)

Restate the monad law of associativity in terms of `flatMap` using `join`, `map` and `unit`.

We first look at the associative law expressed in terms of `flatMap` based on the previously established premise:

```
flatMap(flatMap(x, f), g) ==
    flatMap(x) { a -> flatMap(f(a), g) }
```

We can substitute `f` and `g` with identity functions, and `x` with a higher kind `y`, to express this differently:

```
        flatMap(flatMap(y, z)) { b -> b } ==
            flatMap(y) { a -> flatMap(z(a)) { b -> b } }

        flatMap(flatMap(y, z)) { it } ==
            flatMap(y) { a -> flatMap(a) { it } }
```

We also know from exercise 11.12 that `join` is a `flatMap` combined with an identity function:

```
        flatMap(join(y)) { it } ==
            flatMap(y) { join(it) }
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    join(join(y)) ==
        flatMap(y) { join(it) }
```

We have also learned in exercise 11.3 that `flatMap` can be expressed as a `map` and `join`, this eliminating the final `flatMap`.

```
join(join(y)) ==
    join(map(y) { join(it) })
```

Lastly, we now substitute occurrences of `join(y)` with `unit(x)`, which in both cases amounts to `Kind<F, A>`

```
join(unit(x)) ==
    join(map(x) { unit(it) })
```

EXERCISE 11.14 (HARD/OPTIONAL)

In your own words, write down an explanation of what the associative law means for `Par` and `Parser`.

For `Par` , the join combinator means something like "make the outer thread wait for the inner one to finish." What this law is saying is that if you have threads starting threads three levels deep, then joining the inner threads and then the outer ones is the same as joining the outer threads and then the inner ones.

For `Parser`, the join combinator is running the outer parser to produce a `Parser`, then running the inner Parser on the remaining input. The associative law is saying, roughly, that only the order of nesting matters, since that's what affects the order in which the parsers are run.

EXERCISE 11.15 (HARD/OPTIONAL)

Explain in your own words what the identity laws are stating in concrete terms for `Gen` and `List`.

The left identity law for `Gen`: The law states that if you take the values generated by `unit(a)` (which are always `a`) and apply `f` to those values, that's exactly the same as the generator returned by `f(a)`.

The right identity law for `Gen`: The law states that if you apply `unit` to the values inside the generator `a`, that does not in any way differ from `a` itself.

The left identity law for `List`: The law says that wrapping a list in a singleton `List` and then flattening the result is the same as doing nothing.

The right identity law for `List`: The law says that if you take every value in a list, wrap each one in a singleton `List`, and then flatten the result, you get the list you started with.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 11.16

Implement `map`, `flatMap` and `unit` as methods on this class, and give an implementation for `Monad<Id>`.

```
data class Id<out A>(val a: A) : IdOf<A> {
    companion object {
        fun <A> unit(a: A): Id<A> = Id(a)
    }

    fun <B> flatMap(f: (A) -> Id<B>): Id<B> = f(this.a)
    fun <B> map(f: (A) -> B): Id<B> = unit(f(this.a))
}

class ForId private constructor() {
    companion object
}

typealias IdOf<A> = Kind<ForId, A>

fun <A> IdOf<A>.fix() = this as Id<A>

val idMonad = object : Monad<ForId> {
    override fun <A> unit(a: A): IdOf<A> =
        Id.unit(a)

    override fun <A, B> flatMap(fa: IdOf<A>, f: (A) -> IdOf<B>): IdOf<B> =
        fa.fix().flatMap { a -> f(a).fix() }

    override fun <A, B> map(fa: IdOf<A>, f: (A) -> B): IdOf<B> =
        fa.fix().map(f)
}
```

EXERCISE 11.17

Now that we have a `State` monad, try it out to see how it behaves. Declare some values of `replicateM`, `map2` and `sequence` with type declarations using the `intMonad` above. Describe how each one behaves under the covers?

```
   val replicateIntState: StateOf<Int, List<Int>> =
       intMonad.replicateM(5, stateA)

   val map2IntState: StateOf<Int, Int> =
       intMonad.map2(stateA, stateB) { a, b -> a * b }

   val sequenceIntState: StateOf<Int, List<Int>> =
       intMonad.sequence(List.of(stateA, stateB))
```

`replicateM` for `State` repeats the same state transition a number of times and returns a list of the results. It's not passing the same starting state many times, but chaining the calls together so that the output state of one is the input state of the next.

`map2` works similarly in that it takes two state transitions and feeds the output state of one to the input of the other. The outputs are not put in a list, but combined with a function `f`.

`sequence` takes an entire list of state transitions and does the same kind of thing as `replicateM`: it feeds the output state of the first state transition to the input state of the next, and so on. The results are accumulated in a list.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE 11.18

Express the laws that you would expect to mutually hold for `getState`, `setState`, `unit`, and `flatMap`?

```
getState<Int>().flatMap { a -> setState(a) } == unit<Int, Unit>(Unit)

setState<Int>(1).flatMap { _ -> getState<Int>() } == unit<Int, Int>(1)
```

EXERCISE 11.19 (HARD)

To cement your understanding of monads, give a monad instance for the `Reader` data type and explain what it means. Also, take some time to answer the following questions:

- what are its primitive operations?
- What is the action of `flatMap`?
- What meaning does it give to monadic functions like `sequence`, `join`, and `replicateM`?
- What meaning does it give to the monadic laws?

```
sealed class ForReader private constructor() {
    companion object
}

typealias ReaderOf<R, A> = Kind2<ForReader, R, A>

typealias ReaderPartialOf<R> = Kind<ForReader, R>

fun <R, A> ReaderOf<R, A>.fix() = this as Reader<R, A>

interface ReaderMonad<R> : Monad<ReaderPartialOf<R>>

data class Reader<R, A>(val run: (R) -> A) : ReaderOf<R, A> {

    companion object {
        fun <R, A> unit(a: A): Reader<R, A> = Reader { a }
    }

    fun <B> map(f: (A) -> B): Reader<R, B> =
        this.flatMap { a: A -> unit<R, B>(f(a)) }

    fun <B> flatMap(f: (A) -> Reader<R, B>): Reader<R, B> =
        Reader { r: R -> f(run(r)).run(r) }

    fun <A> ask(): Reader<R, R> = Reader { r -> r }
}

fun <R> readerMonad() = object : ReaderMonad<R> {
    override fun <A> unit(a: A): ReaderOf<R, A> =
        Reader { a }

    override fun <A, B> flatMap(
        fa: ReaderOf<R, A>,
        f: (A) -> ReaderOf<R, B>
    ): ReaderOf<R, B> =
        fa.fix().flatMap { a -> f(a).fix() }
}
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

SIDEBAR

The action of `flatMap` here is to pass the `r` argument along to both the outer `Reader` and also to the result of `f`, the inner `Reader`. Similar to how `State` passes along a state, except that in `Reader` the "state" is read-only.

The meaning of `sequence` here is that if you have a list of functions, you can turn it into a function that takes one argument and passes it to all the functions in the list, returning a list of the results.

The meaning of `join` is simply to pass the same value as both arguments to a binary function.

The meaning of `replicateM` is to apply the same function a number of times to the same argument, returning a list of the results. Note that if this function is pure, (which it should be), this can be exploited by only applying the function once and replicating the result instead of calling the function many times. This means the `Reader` monad can override `replicateM` to provide a very efficient implementation.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Appendix C. Expansive exercises

C.1 Introduction

This appendix features a few exercises that appear in part 2 of the original *Functional Programming in Scala* book, especially chapter 8 on property-based testing. The exercises we moved here are hard, optional, and target a level of reflection and modelling that goes beyond the scope of their respective chapters and are therefore qualified as *open-ended*.

We suggest hints and answers to some of these exercises in this same appendix, but we encourage you to tackle them first, keeping hints and solutions in case you get stuck while working on them.

C.2 Exercises

EXERCISE C.1

A `check` property is easy to prove conclusively because the test just involves evaluating the `Boolean` argument. But some `forAll` properties can be proved as well. For instance, if the domain of the property is `Boolean`, then there are really only two cases to test. If a property `forAll(p)` passes for both `p(true)` and `p(false)`, then it is proved. Some domains (like `Boolean` and `Byte`) are so small that they can be exhaustively checked. And with sized generators, even infinite domains can be exhaustively checked up to the maximum size. Automated testing is very useful, but it's even better if we can *automatically prove our code correct*. Modify our library to incorporate this kind of exhaustive checking of finite domains and sized generators. This is less of an exercise and more of an extensive, open-ended design project.

EXERCISE C.2

We want to generate a function that *uses its argument* in some way to select which `Int` to return. Can you think of a good way of expressing this? This is a very open-ended and challenging design exercise. See what you can discover about this problem and if there's a nice general solution that you can incorporate into the library we've developed so far.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

EXERCISE C.3

You're strongly encouraged to venture out and try using the library we've developed! See what else you can test with it, and see if you discover any new idioms for its use or perhaps ways it could be extended further or made more convenient. Here are a few ideas to get you started:

- Write properties to specify the behavior of some of the other functions we wrote for `List` and `Stream`, for instance, `take`, `drop`, `filter`, and `unfold`.
- Write a sized generator for producing the `Tree` data type defined in chapter 3, and then use this to specify the behavior of the `fold` function we defined for `Tree`. Can you think of ways to improve the API to make this easier?
- Write properties to specify the behavior of the `sequence` function we defined for `Option` and `Either`.

C.3 Hints

EXERCISE C.1

You will need to add to the representation of `Gen`. For example, `Gen<Int>` should be capable of generating random integers as well as generating a stream of all the integers from `Int.MIN_VALUE` to `Int.MAX_VALUE`. You may want to have the behavior depend on how many test cases were requested.

EXERCISE C.2

If we are just looking at the random case, one way to have the generated `Int` depend on the `String` might be to set the seed of a new random number generator to be equal to the hashCode of the given input `String`.

C.4 Answers

C.4.1 Property based testing

EXERCISE C.1

A detailed answer is to be found in the file `Exhaustive.kt` in the code accompanying this appendix.

EXERCISE C.2

Let's start by looking at the signature of our motivating example, generating a function from `(String) -> Int` given a `Gen<Int>`:

```
fun genStringInt(g: Gen<Int>): Gen<(String) -> Int> = TODO()
```

And let's generalize this a bit to not be specialized to `Int`, because that would let us cheat a bit (by, say, returning the `hashCode` of the input `String`, which just so happens to be an `Int`).

```
fun <A> genStringFn(g: Gen<A>): Gen<(String) -> A> = TODO()
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

We've already ruled out just returning a function that ignores the input `String`, since that's not very interesting! Instead, we want to make sure we *use information from* the input `String` to influence what `A` we generate. How can we do that? Well, the only way we can have any influence on what value a `Gen` produces is to modify the `RNG` value it receives as input:

Recall our definition of `Gen`:

```
data class Gen<out A>(val sample: State<RNG, A>)
```

Just by following the types, we can start writing:

```
fun <A> genStringFn(g: Gen<A>): Gen<(String) -> A> = Gen {
    State { (rng: RNG) -> ??? }
}
```

where `???` has to be of type `((String) -> A, RNG)`, and moreover, we want the `String` to somehow affect what `A` is generated. We do that by modifying the seed of the `RNG` before passing it to the `Gen<A>` sample function. A simple way of doing this is to compute the hash of the input string, and mix this into the `RNG` state before using it to produce an `A`:

```
fun <A> genStringFn(g: Gen<A>): Gen<(String) -> A> = Gen {
    State { (rng: RNG) ->
        {
            // we still use `rng` to produce a seed, so we get a new function each time
            val (seed, rng2) = rng.nextInt()
            val f: (String) -> A = s: String -> g.sample.run(
                    SimpleRNG(seed.toLong() xor s.hashCode().toLong())).first
            return (f, rng2)
        }
    }
}
```

More generally, any function which takes a `String` and an `RNG` and produces a new `RNG` could be used. Here, we're computing the `hashCode` of the `String` and then XOR'ing it with a seed value to produce a new `RNG`. We could just as easily take the length of the `String` and use this value to perturn our RNG state, or take the first 3 characters of the string. The choices affect what sort of function we are producing:

- If we use `hashCode` to perturb the `RNG` state, the function we are generating uses all the information of the `String` to influence the `A` value generated. Only input strings that share the same `hashCode` are guaranteed to produce the same `A`.
- If we use the `length`, the function we are generating is using only some of the information of the `String` to influence the `A` being generated. For all input strings that have the same length, we are guaranteed to get the same `A`.

The strategy we pick depends on what functions we think are realistic for our tests. Do we want functions that use all available information to produce a result, or are we more interested in functions that use only bits and pieces of their input? We can wrap the policy up in an interface:

```
interface Cogen<in A> {
```

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

```
    fun sample(a: A, rng: RNG): RNG
}
```

As an exercise, try implementing a generalized version of `genStringFn`.

```
fun <A, B> fn(cin: Cogen<A>, cout: Gen<B>): Gen<(A) -> B> = TODO()
```

You can pattern the implementation after `genStringFn`. Just follow the types!

One problem with this approach is reporting test case failures back to the user. In the event of a failure, all the user will see is that for some opaque function, the property failed, which isn't very enlightening. There's been work in the Haskell library [QuickCheck](www.cse.chalmers.se/~rjmh/QuickCheck/manual.html) to be able to report back to the user and even *shrink* down the generated functions to the simplest form that still falsifies the property. See [this talk on shrinking and showing functions](www.youtube.com/watch?v=CH8UQJiv9Q4).

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

Notes

1. More can be learned about Arrow from their website at arrow-kt.io/

2. *Procedure* is often used to refer to some parameterized chunk of code that may have side effects.

3. pronounced like “ripple,” but with an e instead of an i

4. We can write while loops by hand in Kotlin, but it’s rarely necessary and considered bad form since it hinders good compositional style.

5. The term optimization is not really appropriate here. An optimization usually connotes some nonessential performance improvement, but when we use tail calls to write loops, we generally rely on their being compiled as iterative loops that don’t consume a call stack frame for each iteration (which would result in a StackOverflowError for large inputs).

6. Technically, all values in Kotlin can be compared for equality (using `==`) and turned into a string representation with `toString()`, and an integer can be generated from a value’s internals using `hashCode()`. But this is something of a wart inherited from Java.

7. Even though it’s a fun puzzle, this isn’t a purely academic exercise. Functional programming in practice involves a lot of fitting building blocks together in the only way that makes sense. The purpose of this exercise is to get practice using higher-order functions, and using Kotlin’s type system to guide your programming.

8. Within the body of this inner function, the outer `a` is still in scope. We sometimes say that the inner function closes over its environment, which includes `a`.

9. This is named after the mathematician Haskell Curry, who discovered the principle. It was independently discovered earlier by Moses Schoenfinkel, but Schoenfinkelization didn’t catch on.

10. discuss.kotlinlang.org/t/destructuring-in-when/2391

11. The type annotation `Nil: List<Int>` is needed here, because otherwise Kotlin infers the `B` type parameter in `foldRight` as `List<Nothing>`.

12. In the standard library, `map` and `flatMap` are methods of `List`.

13. kotlinlang.org/api/latest/jvm/stdlib/kotlin.collections/-list/index.html

14. In Kotlin, the implementations of an ADT are restricted by making the base class sealed. This prevents altering the class hierarchy by introducing user-defined implementations

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

15. The naming is not coincidental. There's a deep connection, beyond the scope of this book, between the "addition" and "multiplication" of types to form an ADT and addition and multiplication of numbers.

16. Arity is the number of arguments or operands that a function or operation in logic, mathematics, and computer science takes.

17. A function may also be partial if it doesn't terminate for some inputs. We won't discuss this form of partiality here, since it's not a recoverable error and there's no question of how best to handle it.

18. In general, we'll use this object-oriented style of syntax where possible for functions that have a single, clear operand (like `List.map`), and the standalone function style otherwise.

19. Perhaps it should have been called `mapFlat`, but that doesn't sound quite as appealing!

20. This is a clear instance where it's not appropriate to define the function in the OO style. This shouldn't be a method on `List` (which shouldn't need to know anything about `Option`), and it can't be a method on `Option`, so it goes in the `Option` companion object.

21. `Either` is also often used more generally to encode one of two possibilities in cases where it isn't worth defining a fresh data type. We'll see some examples of this throughout the book.

22. With program traces like these, it's often more illustrative to not fully trace the evaluation of every subexpression. In this case, we've omitted the full expansion of `List.of(1,2,3,4).map { it + 10 }`. We could "enter" the definition of map and trace its execution step by step, but we chose to omit this level of detail for the sake of simplicity.

23. In fact, the type `() -> A` is a syntactic alias for the type `Function<A>`.

24. kotlinlang.org/docs/reference/delegated-properties.html

25. Recall that Kotlin uses subtyping to represent data constructors, but we almost always want to infer `Stream` as the type, not `Cons` or `Empty`. Making smart constructors that return the base type is a common trick.

26. This definition of `exists`, though illustrative, isn't stack-safe if the stream is large and all elements test `false`.

27. It's possible to define a stack-safe version of `forAll` using an ordinary recursive loop.

28. In Kotlin, the `Int` type is a 32-bit signed integer, so this stream will switch from positive to negative values at some point, and will repeat itself after about four billion elements.

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

29. Using `unfold` to define `constant` and `ones` means that we don't get sharing as in the recursive definition `fun ones(): Stream<Int> = Stream.cons({ 1 }, { ones() })`. The recursive definition consumes constant memory even if we keep a reference to it around while traversing it, while the `unfold`-based implementation does not. Preserving sharing isn't something we usually rely on when programming with streams, since it's extremely delicate and not tracked by the types. For instance, sharing is destroyed when calling even `xs.map { x -> x }`.

30. Actually, pseudo-random, but we'll ignore this distinction for our purposes.

31. Kotlin API link: bit.ly/35MLFhz

32. Recall that `Pair<A, B>` is the type of two-element tuples, and given `p: Pair<A, B>`, you can use `p.first` to extract the `A` and `p.second` to extract the `B`.

33. An efficiency loss comes with computing next states using pure functions, because it means we can't actually mutate the data in place. Here, it's not really a problem since the state is just a single `Long` that must be copied. This loss of efficiency can be mitigated by using efficient purely functional data structures. It's also possible in some cases to mutate the data in place without breaking referential transparency, which we'll talk about in part 4 of this book.

34. We'll use the term *logical thread* somewhat informally to mean a computation that runs concurrently with the main execution thread of our program. There need not be a one-to-one correspondence between logical threads and OS threads. We may have a large number of logical threads mapped onto a smaller number of OS threads via thread pooling, for instance.

35. We do mean algebra in the mathematical sense of one or more sets, together with a collection of functions operating on objects of these sets, and a set of *axioms*. Axioms are statements assumed true from which we can derive other *theorems* that must also be true. In our case, the sets are particular types like `Par<A>` and `List<Par<A>>`, and the functions are operations like `map2`, `unit`, and `sequence`.

36. docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/concurrent/Executor.html

37. This last way of defining new laws is probably the weakest, since it can be too easy to just have the laws reflect the implementation, even if the implementation is buggy or requires all sorts of unusual side conditions that make composition difficult.

38. This is the same usage of "domain" as the domain of a function (en.wikipedia.org/wiki/Domain_of_a_function)—generators describe possible inputs to functions we'd like to test. Note that we'll also still sometimes use "domain" in the more colloquial sense, to refer to a subject or area of interest, for example, "the domain of functional parallelism" or "the error-handling domain."

39. www.scalacheck.org

40. hackage.haskell.org/package/QuickCheck

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion
Licensed to John Edwards <john.edwards@jfinancial.co.uk>

41. Arrow provides a `forAll` extension method for `List` and `Sequence` with the signature `fun <A> List<A>.forAll(f: (A) -> Boolean>): Boolean`.

42. en.wikipedia.org/wiki/JSON

43. en.wikipedia.org/wiki/Yacc

44. www.antlr.org/

45. The time `List.size()` takes depends on the implementation. For example, if the number of items is stored in an integer, `size()` will take constant time.

46. en.wikipedia.org/wiki/JSON

47. json.org

48. The `copy` method comes for free with any `data class`. It returns a copy of the object, but with one or more attributes modified. If no new value is specified for a field, it will have the same value as in the original object. This uses the same mechanism as default parameters in Kotlin

49. The Greek prefix *endo-* means *within*, in the sense that an endofunction's codomain is within its domain.

50. *Homomorphism* comes from Greek, *homo* meaning "same" and *morphe* meaning "shape."

51. The semigroup defines the ability to combine two values of the same type, the monoid adds to this by providing an empty or `nil` value.

52. Our decision to call the type constructor argument `F` here was arbitrary. We could have called this argument `Foo`, `w00t`, or `Blah2`, though by convention, we usually give type constructor arguments one-letter uppercase names, such as `F`, `G`, and `H`, or sometimes `M` and `N`, or `P` and `Q`.

53. The Greek prefix *endo-* means *within*, in the sense that an endofunction's codomain is within its domain.

54. The Greek prefix *endo-* means *within*, in the sense that an endofunction's codomain is within its domain.

Index Terms

checked exceptions
exception handling (checked exceptions)

©Manning Publications Co. To comment go to liveBook
https://livebook.manning.com/#!/book/functional-programming-with-kotlin/discussion