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`.

```
def flatMap[A,B](f: Rand[A])(g: A => Rand[B]): Rand[B]
```

`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`.

We can now revisit our example from the beginning of this chapter. 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:

```
def rollDie: Rand[Int] = nonNegativeLessThan(6)
```

If we test this function with various `RNG` states, we'll pretty soon find an `RNG` that causes this function to return 0:

```
scala> val zero = rollDie(SimpleRNG(5))._1
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 it's been used.

Fixing the bug is trivial:

```
def rollDie: Rand[Int] = map(nonNegativeLessThan(6))(_ + 1)
```

6.5 A general state action data type

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 we can give it a more general signature:

```
def boolean: Gen[Boolean]
def listOfN[A](n: Int, g: Gen[A]): Gen[List[A]]
```

Generates lists of length `n` using the generator `g`

As we discussed in chapter 7, we're interested in understanding what operations are *primitive* and what operations are *derived*, and in finding a small yet expressive set of primitives. A good way to explore what is expressible 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 some ideas:

- If we can generate a single `Int` in some range, do we need a new primitive to generate an `(Int,Int)` pair in some range?
- Can we produce a `Gen[Option[A]]` from a `Gen[A]`? What about a `Gen[A]` from a `Gen[Option[A]]`?
- Can we generate strings somehow using our existing primitives?

The importance of play

You don't have to wait around for a concrete example to force exploration of the design space. In fact, if you rely exclusively on concrete, obviously useful or important examples to design your API, you'll often miss out on aspects of the design space and generate APIs with ad hoc, 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 *play*. Don't try to solve important problems or produce useful functionality. Not right away. Just experiment with different representations, primitives, and operations, let questions naturally arise, and explore whatever piques your curiosity. ("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?") There's 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[(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

Using these primitives, we can express repetition and nonempty 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 what *can't* we express yet?

Suppose we want to parse a single digit, like `'4'`, followed by *that many* `'a'` characters (this sort of problem should feel familiar from previous chapters). Examples of valid input are `"0"`, `"1a"`, `"2aa"`, `"4aaaa"`, and so on. This is an example of a context-sensitive grammar. It can't be expressed with `product` because our choice of the second parser *depends on* the result of the first (the second parser depends on its context). We want to run the first parser, and then do a `listOfN` using the number extracted from the first parser's result. Can you see why `product` can't express this?

This progression might feel familiar to you. In past chapters, we encountered similar expressiveness limitations and dealt with it by introducing a new primitive, `flatMap`. Let's introduce that here (and we'll add an alias to `ParserOps` so we can write parsers using for-comprehensions):

```
def flatMap[A,B](p: Parser[A])(f: A => Parser[B]): Parser[B]
```

Can you see how this signature implies an ability to sequence parsers where each parser in the chain depends on the output of the previous one?

EXERCISE 9.6

Using `flatMap` and any other combinators, write the context-sensitive parser we couldn't express earlier. To parse the digits, you can make use of a new primitive, `regex`, which promotes a regular expression to a `Parser`.[10] In Scala, a string `s` can be promoted to a `Regex` object (which has methods for matching) using `s.r`, for instance, `"[a-zA-Z_][a-zA-Z0-9_]*".r`.

```
implicit def regex(r: Regex): Parser[String]
```

EXERCISE 9.7

Implement `product` and `map2` in terms of `flatMap`.

EXERCISE 9.8

`map` is no longer primitive. Express it in terms of `flatMap` and/or other combinators.

[10] In theory this isn't necessary; we could write out `"0" | "1" | ... "9"` to recognize a single digit, but this isn't likely to be very efficient.

These type signatures differ only in the concrete data type (`Gen`, `Parser`, or `Option`). We can capture as a Scala trait the idea of "a data type that implements `map`":

```
trait Functor[F[_]] {
  def map[A,B](fa: F[A])(f: A => B): F[B]
}
```

Here we've parameterized `map` on the type constructor, `F[_]`, much like we did with `Foldable` in the previous chapter.[2] Instead of picking a particular `F[_]`, like `Gen` or `Parser`, the `Functor` trait is parametric in the choice of `F`. Here's an instance for `List`:

```
val listFunctor = new Functor[List] {
  def map[A,B](as: List[A])(f: A => B): List[B] = as map f
}
```

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? As we did in several places throughout this book, we can discover useful functions just by *playing* with the operations of the interface, in a purely algebraic way. You may want to pause here to see what (if any) useful operations you can define only in terms of `map`.

Let's look at one example. If we have `F[(A, B)]` where `F` is a functor, we can "distribute" the `F` over the pair to get `(F[A], F[B])`:

```
trait Functor[F[_]] {
  ...
  def distribute[A,B](fab: F[(A, B)]): (F[A], F[B]) =
    (map(fab)(_._1), map(fab)(_._2))
}
```

We wrote this just by 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[(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:

```
def codistribute[A,B](e: Either[F[A], F[B]]): F[Either[A, B]] =
  e match {
    case Left(fa) => map(fa)(Left(_))
    case Right(fb) => map(fb)(Right(_))
  }
```

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 actually have.

---

[2] Recall that a *type constructor* is applied to a type to produce a 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 to `String` to yield `Parser[String]`.

Listing 15.1 Counting line numbers in imperative style

```
def linesGt40k(filename: String): IO[Boolean] = IO {
  val src = io.Source.fromFile(filename)
  try {
    var count = 0
    val lines: Iterator[String] = src.getLines
    while (count <= 40000 && lines.hasNext) {
      lines.next
      count += 1
    }
    count > 40000
  }
  finally src.close
}
```

`scala.io.Source` has convenience functions for reading from external sources like files.

Obtain a stateful `Iterator` from the `Source`.

Has the side effect of advancing to the next element.

We can then *run* this `IO` action with `unsafePerformIO(linesGt40k("lines.txt"))`, where `unsafePerformIO` is a side-effecting method that takes `IO[A]`, returning `A` and actually performing the desired effects (see section 13.6.1).

Although this code uses low-level primitives like a `while` loop, a mutable `Iterator`, and a `var`, there are some *good* things about it. First, it's *incremental*—the entire file isn't loaded into memory up front. Instead, lines are fetched from the file only when needed. If we didn't buffer the input, we could keep as little as a single line of the file in memory at a time. It also terminates early, as soon as the answer is known.

There are some bad things about this code, too. For one, we have to remember to close the file when we're done. This might seem obvious, but if we forget to do this, or (more commonly) if we close the file outside of a `finally` block and an exception occurs first, the file will remain open.[1] This is called a *resource leak*. A file handle is an example of a scarce resource—the operating system can only have a limited number of files open at any given time. If this task were part of a larger program—say we were scanning an entire directory recursively, building up a list of all files with more than 40,000 lines—our larger program could easily fail because too many files were left open.

We want to write programs that are *resource-safe*—they should close file handles as soon as they're no longer needed (whether because of normal termination or an exception), and they shouldn't attempt to read from a closed file. Likewise for other resources like network sockets, database connections, and so on. Using `IO` directly can be problematic because it means our programs are entirely responsible for ensuring their own resource safety, and we get no help from the compiler in making sure that they do this. It would be nice if our library would ensure that programs are resource-safe by construction.

But even aside from the problems with resource safety, there's something unsatisfying about this code. It entangles the high-level algorithm with low-level concerns about iteration and file access. Of course we have to obtain the elements from some

[1] The JVM will actually close an `InputStream` (which is what backs a `scala.io.Source`) when it's garbage collected, but there's no way to guarantee this will occur in a timely manner, or at all! This is especially true in generational garbage collectors that perform "full" collections infrequently.

We use a helper function, `kill`—it feeds the `Kill` exception to the outermost `Await` of a `Process` but ignores any of its remaining output.

Listing 15.8 `kill` helper function

```
@annotation.tailrec
final def kill[O2]: Process[F,O2] = this match {
  case Await(req,recv) => recv(Left(Kill)).drain.onHalt {
    case Kill => Halt(End)
    case e => Halt(e)
  }
  case Halt(e) => Halt(e)
  case Emit(h, t) => t.kill
}

final def drain[O2]: Process[F,O2] = this match {
  case Halt(e) => Halt(e)
  case Emit(h, t) => t.drain
  case Await(req,recv) => Await(req, recv andThen (_.drain))
}
```

We convert the `Kill` exception back to normal termination.

Note that `|>` is defined for any `Process[F,O]` type, so this operation works for transforming a `Process1` value, an effectful `Process[IO,O]`, and the two-input `Process` type we'll discuss next.

With `|>`, we can add convenience functions on `Process` for attaching various `Process1` transformations to the output. For instance, here's `filter`, defined for any `Process[F,O]`:

```
def filter(f: O => Boolean): Process[F,O] =
  this |> Process.filter(f)
```

We can add similar convenience functions for `take`, `takeWhile`, and so on. See the chapter code for more examples.

15.3.4 Multiple input streams

Imagine if we wanted to "zip" together two files full of temperatures in degrees Fahrenheit, `f1.txt` and `f2.txt`, add corresponding temperatures together, convert the result to Celsius, apply a five-element moving average, and output the results one at a time to `celsius.txt`.

We can address these sorts of scenarios with our general `Process` type. Much like effectful sources and `Process1` were just specific instances of our general `Process` type, a `Tee`, which combines two input streams in some way,[9] can also be expressed as a `Process`. Once again, we simply craft an appropriate choice of `F`:

```
case class T[I,I2]() {
  sealed trait f[X] { def get: Either[I => X, I2 => X] }
  val L = new f[I] { def get = Left(identity) }
```

[9] The name *Tee* comes from the letter *T*, which approximates a diagram merging two inputs (the top of the T) into a single output.

character), followed by the name of the member, as in `MyModule.abs(-42)`. To use the `toString` member on the object `42`, we'd use `42.toString`. The implementations of members within an object can refer to each other unqualified (without prefixing the object name), but if needed they have access to their enclosing object using a special name: `this`.[2]

Note that even an expression like `2 + 1` is just calling a member of an object. In that case, what we're calling is the `+` member of the object `2`. It's really syntactic sugar for the expression `2.+(1)`, which passes `1` as an argument to the method `+` on the object `2`. Scala has no special notion of *operators*. It's simply the case that `+` is a valid method name in Scala. Any method name can be used infix like that (omitting the dot and parentheses) when calling it with a single argument. For example, instead of `MyModule.abs(42)` we can say `MyModule abs 42` and get the same result. You can use whichever you find more pleasing in any given case.

We can bring an object's member into scope by *importing* it, which allows us to call it unqualified from then on:

```
scala> import MyModule.abs
import MyModule.abs

scala> abs(-42)
res0: 42
```

We can bring *all* of an object's (nonprivate) members into scope by using the underscore syntax:

```
import MyModule._
```

2.4 Higher-order functions: passing functions to functions

Now that we've covered the basics of Scala's syntax, we'll move on to 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. 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] Note that in this book, we'll use the term *function* to refer more generally to either so-called standalone functions like `sqrt` or `abs`, or members of some class, including methods. When it's clear from the context, we'll also use the terms *method* and *function* interchangeably, since what matters is not the syntax of invocation (`obj.method(12)` vs. `method(obj, 12)`, but the fact that we're talking about some parameterized block of code.

```
def if2[A](cond: Boolean, onTrue: => A, onFalse: => A): A =
  if (cond) onTrue else onFalse
```

The arguments we'd like to pass unevaluated have an arrow `=>` immediately before their type. In the body of the function, we don't need to do anything special to evaluate an argument annotated with `=>`. We just reference the identifier as usual. Nor do we have to do anything special to call this function. We just use the normal function call syntax, and Scala takes care of wrapping the expression in a thunk for us:

```
scala> if2(false, sys.error("fail"), 3)
res2: Int = 3
```

With either 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, Scala won't (by default) cache the result of evaluating an argument:

```
scala> def maybeTwice(b: Boolean, i: => Int) = if (b) i+i else 0
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice(true, { println("hi"); 1+41 })
hi
hi
x: Int = 84
```

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 using the `lazy` keyword:

```
scala> def maybeTwice2(b: Boolean, i: => Int) = {
     |   lazy val j = i
     |   if (b) j+j else 0
     | }
maybeTwice: (b: Boolean, i: => Int)Int

scala> val x = maybeTwice2(true, { println("hi"); 1+41 })
hi
x: Int = 84
```

Adding the `lazy` keyword to a `val` declaration will cause Scala to delay evaluation of the right-hand side of that `lazy val` declaration until it's first referenced. It will also cache the result so that subsequent references to it don't trigger repeated evaluation.

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.