Once you're confident that you have the correct function signature, sketch the function body here:

Solution

In this case, because `ifBothTrue` takes two test conditions followed by a block of code, and it doesn't return anything, its signature looks like this:

```
def ifBothTrue(test1: => Boolean)(test2: => Boolean)(codeBlock: => Unit): Unit = ???
```

Because the code block should only be run if both test conditions are `true`, the complete function should be written like this:

```
def ifBothTrue(test1: => Boolean)(test2: => Boolean)(codeBlock: => Unit): Unit = {
    if (test1 && test2) {
        codeBlock
    }
}
```

You can test `ifBothTrue` with code like this:

```
val age = 19
val numAccidents = 0
ifBothTrue(age > 18)(numAccidents == 0) { println("Discount!") }
```

This also works:

```
ifBothTrue(2 > 1)(3 > 2)(println("hello"))
```

29

Recursion: Introduction

As you may have noticed from this book's index, you're about to jump into a series of lessons on recursive programming. I separated this text into a series of small lessons to make the content easier to read initially, and then easier to refer to later.

Please note that some of these lessons may be overkill for some people. This is, after all, the first draft of this book, and I'm trying to find the best ways to teach recursive programming. I start by reviewing the `List` class, then show a straightforward, "Here's how to write a recursive function" lesson. After that I add a few more lessons to explain recursion in different ways.

If at any point you feel like you understand how to write recursive functions, feel free to skip any or all of these lessons. You can always come back to them later if you need to.

of generic type `A`, and transforms those parameters into a potentially different type `B`:

```
case class Transform2ParamsTo1Param[A, B](fun: (A, A) => B)
```

To be clear, here's the FIP signature by itself:

```
fun: (A, A) => B
```

Now I can write code like this to create an instance of `Transform2ParamsTo1Param`:

```
val x = Transform2ParamsTo1Param { (a: String, b: String) =>
    a.length + b.length
}
```

Then I can call `fun` on `x` like this:

```
// prints "6"
println(x.fun("foo", "bar"))
```

Because `Transform2ParamsTo1Param` defines its function input parameter `fun` to take generic types, I can also write code like this to take two `Int` values and return an `Int`:

```
val y = Transform2ParamsTo1Param { (a: Int, b: Int) =>
    a + b
}

// prints "3"
println(y.fun(1, 2))
```

While this might seem a little unusual if you come from an OOP background, this is just another example of a class that takes a function input parameter, i.e., an example of passing functions around.

> If code like this isn't comfortable right now, fear not, it wasn't comfortable to me initially either. In my experience, I never got comfortable with it until I started using the technique in my own code. (Pro tip: Write

- `match` expressions, or more rarely,
- `getOrElse`

Let's see what we can do with those.

Adding `x` *and* `y` *with* `match` *expressions*

If you try to add `x` and `y` using `match` expressions, you end up with code like this:

```
val sum = x match {
    case None => {
        y match {
            case None => {
                0
            }
            case Some(i) => {
                i
            }
        }
    }
    case Some(i) => {
        y match {
            case None => {
                i
            }
            case Some(j) => {
                i + j
            }
        }
    }
```

I have to be honest: that code is so ugly I didn't even both to see if it would compile. I definitely don't want to use this approach.

63

Starting to Glue Functions Together

As I mentioned at the beginning of this book, writing pure functions isn't hard. Just make sure that output depends only on input, and you're in good shape.

The hard part of functional programming involves how you glue together all of your pure functions to make a complete application. Because this process feels like you're writing "glue" code, I refer to this process as "gluing," and as I learned, a more technical term for this is called "binding." This process is what the remainder of this book is about.

> As you might have guessed from the emphasis of the previous lessons, in Scala/FP this binding process involves the use of `for` expressions.

Life is good when the output of one function matches the input of another

To set the groundwork for where we're going, let's start with a simplified version of a problem you'll run into in the real world.

Imagine that you have two functions named `f` and `g`, and they both a) take an `Int` as an input parameter, and b) return an `Int` as their result. Therefore, their function signatures look like this:

```
def f(a: Int): Int = ???
def g(a: Int): Int = ???
```

A nice thing about these functions is that because the output of `f` is an `Int`, it perfectly matches the input of `g`, which takes an `Int` parameter. This is shown visually in Figure 63.1.

```
object Golfing3 extends App {

    case class GolfState(distance: Int)

   def swing(distance: Int): State[GolfState, Int] = State { (s: GolfState) =>
        val newAmount = s.distance + distance
        (GolfState(newAmount), newAmount)
    }

    val stateWithNewDistance: State[GolfState, Int] = for {
        _             <- swing(20)
        _             <- swing(15)
        totalDistance <- swing(0)
    } yield totalDistance

    // initialize a `GolfState`
    val beginningState = GolfState(0)

    // run/execute the effect.
    // `run` is like `unsafeRunSync` in the Cats `IO` monad.
   val result: (GolfState, Int) = stateWithNewDistance.run(beginningState)

    println(s"GolfState:      ${result._1}")  //GolfState(35)
    println(s"Total Distance: ${result._2}")  //35

}
```

I'll explain this code in the remainder of this lesson.

GolfState

First, as with the previous lesson, I use a `case` class to model the distance of each stroke:

```
case class GolfState(distance: Int)
```

```
val price = pizza3.getPrice(
    toppingPrices,
    crustSizePrices,
    crustTypePrices
)
```

Notice that in this line:

```
val pizza2 = pizza1.addTopping(Pepperoni)
```

the `Pepperoni` topping is added to whatever toppings are in the `pizza1` reference to create a new `Pizza` instance named `pizza2`. Following the FP model, `pizza1` isn't mutated, it's just used to create a new instance with the updated data.

Are the functions pure?

In this example, because `getPrice` doesn't take any input parameters, by my own definition it's not a strictly pure function. You can't say, "Output depends only on input," because there appears to be no input.

That being said, what's really going on with these method calls is that they receive an implicit `this` reference, so under the covers they really look like this:

```
val pizza2 = pizza1.addTopping(this, Pepperoni)
val pizza3 = pizza2.updateCrustType(this, ThickCrustType)

val price = pizza3.getPrice(
    this,
    toppingPrices,
    crustSizePrices,
    crustTypePrices
)
```

In that regard, these methods on the `Pizza` class are as pure as methods like `map` and `filter` on the `List` class.

Note: Prefer Try over Option for I/O functions

I prefer to use `Try` for file and network I/O functions because I usually want to know what the exception was when a function fails. `Try` gives me the exception when there is a problem, but `Option` just returns `None`, which doesn't tell me what the actual problem was.

Source code

The source code for this lesson is available at the following URL:

- github.com/alvinj/FPIOMonadNotReallyUsed

Discussion

To demonstrate this situation, let's look at a short example. First, imagine that you already have this `readTextFileAsString` function, which returns `Try[String]`:

```
def readTextFileAsString(filename: String): Try[String] =
    Try {
        val lines = using(io.Source.fromFile(filename)) { source =>
            (for (line <- source.getLines) yield line).toList
        }
        lines.mkString("\n")
    }
```

As I showed in the "Functional Error Handling lesson," one way to use functions that return `Try` is like this:

```
val passwdFile = readTextFileAsString("/etc/passwdFoo")
passwdFile match {
    case Success(s) => println(s)
    case Failure(e) => println(e)
}
```

the three futures, which are running on other threads. This isn't usually a problem in the real world, but it's a problem for little demos like this.

Similar to the real world

While that example doesn't do much and you know up front long it takes for each `Future` to complete, it's remarkably similar to code that you'll use in the real world. For example, if you call a function named `Cloud.executeLongRunningTask()` to get an `Int` result that will take an indeterminate amount of time, you'll still construct the future in the same way:

```
val task: Future[Int] = Future {
    Cloud.executeLongRunningTask(a, b, c)
}
```

Then, whenever the future is finished, you'll also use a method like `onComplete` to process the result:

```
task.onComplete {
    case Success(value) => outputTheResult(value)
    case Failure(e) => outputTheError(e)
}
```

So although the code in the curly braces of the f1/f2/f3 example doesn't do much, the use of futures in that example follows the same pattern that you'll use in the real world:

- Construct one or more futures to run tasks off of the main thread
- If you're using multiple futures to yield a single result, combine the futures in a `for` expression
- Use a callback method like `onComplete` to process the final result

4) `for/yield` with two `Option` generators

This `for` expression:

```
val x = for {
    i <- Option(1)
    j <- Option(2)
} yield i * j
```

translates to this code:

```
val x = Option(1).flatMap { i =>
    Option(2).map { j =>
        i * j
    }
}
```

Notes

- as in the previous example, two generators in `for/yield` becomes `flatMap/map`. It doesn't matter if they class in the `for` expression is a `List` or an `Option`.