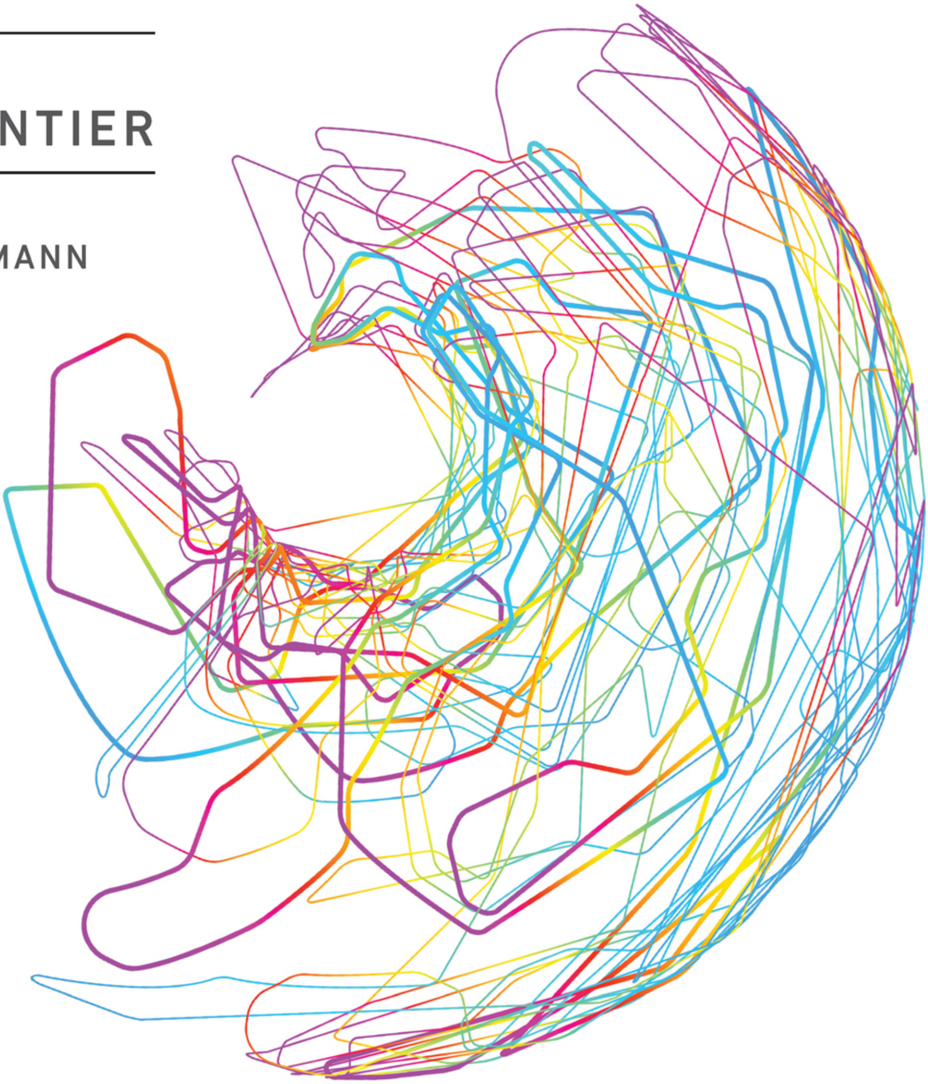
FUNCTIONAL AND CONCURRENT PROGRAMMING

Core Concepts and Features

MICHEL
CHARPENTIER

Foreword by
CAY HORSTMANN



Functional and Concurrent Programming

A limitation of this function, however, is that you can only search for a target if you already have a value equal to that target. For instance, you cannot search a list of temperatures for a value greater than 90. Of course, you can easily write another function for that:

Scala
```scala
def findGreaterThan90(list: List[Int]): Option[Int] = list match
   case Nil    => None
   case h :: t => if h > 90 then Some(h) else findGreaterThan90(t)

findGreaterThan90(temps) // Some(91)
```

But what if you need to search for a temperature greater than 80 instead? You can write another function, in which an integer argument replaces the hardcoded value 90:

Scala
```scala
def findGreaterThan(list: List[Int], bound: Int): Option[Int] = list match
   case Nil    => None
   case h :: t => if h > bound then Some(h) else findGreaterThan(t, bound)

findGreaterThan(temps, 80) // Some(88)
```

This is better, but the new function still cannot be used to search for a temperature *less* than 90, or for a string that ends with `"a"`, or for a project with identity 12345.

You will notice that functions `find`, `findGreaterThan90`, and `findGreaterThan` are strikingly similar. The algorithm is the same in all three cases. The only part of the implementation that changes is the test in the `if-then-else`, which is `h == target` in the first function, `h > 90` in the next, and `h > bound` in the third.

It would be nice to write a generic function `find` parameterized by a search criterion. Criteria such as "to be greater than 90" or "to end with `"a"`" or "to have identity 12345" could then be used as arguments. To implement the `if-then-else` part of this function, you would apply the search criterion to the head of the list to produce a Boolean value. In other words, you need the search criterion to be a *function* from `A` to `Boolean`.

Such a function `find` can be written. It takes another function as an argument, named `test`:

Scala
```scala
def find[A](list: List[A], test: A => Boolean): Option[A] = list match
   case Nil    => None
   case h :: t => if test(h) then Some(h) else find(t, test)
```

Listing 9.2: Recursive implementation of higher-order function `find`.

The type of argument `test` is `A => Boolean`, which in Scala denotes functions from `A` to `Boolean`. As a function, `test` is applied to the head of the list `h` (of type `A`), and produces a value of type `Boolean` (used as the `if` condition).

You can use this new function **find** to search a list of temperatures for a value greater than 90 by first defining the “greater than 90” search criterion as a function:

Scala

```
def greaterThan90(x: Int): Boolean = x > 90
find(temps, greaterThan90) // Some(91)
```

In this last expression, you do not invoke function `greaterThan90` on an integer argument. Instead, you use the function itself as an argument to `find`. To search for a project with identity 12345, simply define a different search criterion:

Scala

```
def hasID12345(project: Project): Boolean = project.id == 12345L
find(projects, hasID12345) // project with identity 12345
```

Because it takes a function as an argument, `find` is said to be a *higher-order* function. Functional programming libraries define many standard higher-order functions, some of which are discussed in Chapter 10. In particular, a method `find` is already defined on Scala’s `List` type. The two searches in the preceding examples can be written as follows:

Scala

```
temps.find(greaterThan90)
projects.find(hasID12345)
```

From now on, code examples in this chapter use the standard method `find` instead of the earlier user-defined function.

Method `find` is a higher-order function because it takes another function as an argument. A function can also be higher-order by returning a value that is a function. For example, instead of implementing `greaterThan90`, you can define a function that builds a search criterion to look for temperatures greater than a given bound:

Scala

```
def greaterThan(bound: Int): Int => Boolean =
   def greaterThanBound(x: Int): Boolean = x > bound
   greaterThanBound
```

Listing 9.3: Example of a function that returns a function; see also Lis. 9.4 and 9.5.

Function `greaterThan` works by first defining a function `greaterThanBound`. This function is not applied to anything but simply returned as a value. Note that `greaterThan` has return type `Int => Boolean`, which denotes functions from integers to Booleans. Given a lower bound `b`, the expression `greaterThan(b)` is a function, which tests whether an integer is greater than `b`. It can be used as an argument to higher-order method `find`:

Scala
```
temps.find(greaterThan(90))
temps.find(greaterThan(80))
```

In a similar fashion, you can define a function to generate search criteria for projects:

Scala
```
def hasID(identity: Long): Project => Boolean =
   def hasGivenID(project: Project): Boolean = project.id == identity
   hasGivenID

projects.find(hasID(12345L))
projects.find(hasID(54321L))
```

9.2 Currying

Functions that return other functions are common in functional programming, and many languages define a more convenient syntax for them:

Scala
```
def greaterThan(bound: Int)(x: Int): Boolean = x > bound
def hasID(identity: Long)(project: Project): Boolean = project.id == identity
```

Listing 9.4: Example of higher-order functions defined through currying.

It might appear as if `greaterThan` is a function of two arguments, `bound` and `x`, but it is not. It is a function of a single argument, `bound`, which returns a function of type `Int => Boolean`, as before; `x` is actually an argument of the function being returned.

Functions written in this style are said to be *curried*.[1] A curried function is a function that consumes its first list of arguments, returns another function that uses the next argument list, and so on. You can read the definition of `greaterThan` as implementing a function that takes an integer argument `bound` and returns another function, which takes an integer argument `x` and returns the Boolean `x > bound`. In other words, the return value of `greaterThan` is the function that maps `x` to `x > bound`.

Functional programming languages rely heavily on currying. In particular, currying can be used as a device to implement all functions as single-argument functions, as in

[1]The concept is named after the logician Haskell Curry, and the words *curried* and *currying* are sometimes capitalized.

languages like Haskell and ML. For instance, we tend to think of addition as a function of two arguments:

Scala
```scala
def plus(x: Int, y: Int): Int = x + y // a function of type (Int, Int) => Int
plus(5, 3)                            // 8
```

However, you can also think of it as a single-argument (higher-order) function:

Scala
```scala
def plus(x: Int)(y: Int): Int = x + y // a function of type Int => (Int => Int)
plus(5)                               // a function of type Int => Int
plus(5)(3)                            // 8
```

Curried functions are so common in functional programming that the `=>` that represents function types is typically assumed to be right-associative: `Int => (Int => Int)` is simply written `Int => Int => Int`. For example, the function

Scala
```scala
def lengthBetween(low: Int)(high: Int)(str: String): Boolean =
   str.length >= low && str.length <= high
```

has type `Int => Int => String => Boolean`. You can use it to produce a Boolean, as in

Scala
```scala
lengthBetween(1)(5)("foo") // true
```

but also to produce other functions:

Scala
```scala
val lengthBetween1AndBound: Int => String => Boolean = lengthBetween(1)
val lengthBetween1and5: String => Boolean            = lengthBetween(1)(5)

lengthBetween1AndBound(5)("foo") // true
lengthBetween1and5("foo")        // true
```

Before closing this section on currying, we should consider a feature that is particular to Scala (although other languages use slightly different tricks for the same purpose).

In Scala, you can call a single-argument function on an expression delimited by curly braces without the need for additional parentheses. So, instead of writing

Scala
```
println({
   val two = 2
   two + two
}) // prints 4
```

you can simply write:

Scala
```
println {
   val two = 2
   two + two
} // prints 4
```

To use this syntax when multiple arguments are involved, you can rely on currying to adapt a multi-argument function into a single-argument function. For instance, the curried variant of function `plus` can be invoked as follows:

Scala
```
plus(5) {
   val two = 2
   two + 1
}
```

This is still value 8, as before.

Many functions and methods are curried in Scala for the sole purpose of benefiting from this syntax. The syntax is introduced here because we will encounter some example uses throughout the book, starting with the next section.

9.3 Function Literals

It would be inconvenient if, to use higher-order functions like `find`, you always had to separately define (and name) argument functions like `hasID12345` and `greaterThan90`. After all, when you call a function on a string or an integer, you don't need to define (and name) the values first. This is because programming languages define a syntax for strings and integer literals, like the `"foo"` and `42` that are sprinkled throughout this book's code illustrations. Similarly, functional programming languages, which rely heavily on higher-order functions, offer syntax for *function literals*, also called *anonymous functions*. The most common form of function literals is lambda expressions, which are often the first thing that comes to mind when you hear that a language has support for functional programming.

In Scala, the syntax for lambda expressions is `(v1: T1, v2: T2, ...) => expr`.[2] This defines a function with arguments `v1`, `v2`, ... that returns the value produced by `expr`. For instance, the following expression is a function, of type `Int => Int`, that adds 1 to an integer:

```scala
(x: Int) => x + 1
```

Function literals can be used to simplify calls to higher-order functions like `find`:

```scala
temps.find((temp: Int) => temp > 90)
projects.find((proj: Project) => proj.id == 12345L)
```

The Boolean functions "to be greater than 90" and "to have identity 12345" are implemented as lambda expressions, which are passed directly as arguments to method `find`.

You can also use function literals as return values of other functions. So, a third way to define functions `greaterThan` and `hasID`, besides using named local functions or currying, is as follows:

```scala
def greaterThan(bound: Int): Int => Boolean = (x: Int) => x > bound
def hasID(identity: Long): Project => Boolean = (p: Project) => p.id == identity
```

Listing 9.5: Example of higher-order functions defined using lambda expressions.

The expression `(x: Int) => x > bound` replaces the local function `greaterThanBound` from Listing 9.3.

Function literals have no name, and usually do not declare their return type. Compilers can sometimes infer the types of their arguments. You could omit argument types in all the examples written so far:

```scala
temps.find(temp => temp > 90)
projects.find(proj => proj.id == 12345L)

def greaterThan(bound: Int): Int => Boolean = x => x > bound
def hasID(identity: Long): Project => Boolean = p => p.id == identity
```

[2]Lambda expressions can also be parameterized by types, though this is a more advanced feature not used in this book. For instance, Listing 2.7 defines a function `first` of type `(A, A) => A`, parameterized by type `A`. It could be written as the lambda expression `[A] => (p: (A, A)) => p(0)`.