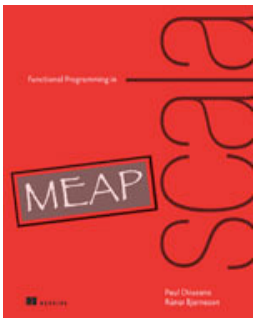
MANNING PUBLICATIONS



Functional Programming in Scala
By Paul Chiusano and Rúnar Bjarnason

Functional programs do not update variables or modify data structures. This raises pressing questions—what sort of data structures we use in functional programming, how do we define them can in Scala, and how do we operate over these data structures? This article, based on chapter 3 of Functional Programming in Scala, explains the concept of a functional data structure and how to define and work with such structures.

You may also be interested in…

Defining Functional Data Structures

A functional data structure is (not surprisingly!) operated on using only pure functions. A pure function may only accept some values as input and yield a value as output. It may not change data in place or perform other side effects. Therefore, functional data structures are immutable. For example, the empty list, (denoted `List()` or `Nil` in Scala) is 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? Somewhat surprisingly, the answer is no. We will return to this issue after examining the definition of what is perhaps the most ubiquitous of functional data structures, the singly-linked list. The definition here is identical in spirit to (though simpler than) the data type defined in Scala's standard List library. This code listing makes use of a lot of new syntax and concepts, so don't worry if not everything makes sense at first—we will talk through it in detail.[1]

Listing 1 Singly-linked lists

```
package fpinscala.datastructures

sealed trait List[+A]      #1
case object Nil extends List[Nothing]     #2
case class Cons[+A](head: A, tail: List[A]) extends List[A]

object List {             #3
  def sum(ints: List[Int]): Int = ints match {      #4
    case Nil => 0
    case Cons(x,xs) => x + sum(xs)
  }

  def product(ds: List[Double]): Double = ds match {
    case Nil => 1.0
    case Cons(0.0, _) => 0.0
    case Cons(x,xs) => x * product(xs)
  }

  def apply[A](as: A*): List[A] =    #5
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
```

[1] Note—the implementations of sum and product here are not tail recursive.

```
  val example = Cons(1, Cons(2, Cons(3, Nil)))    #6
  val example2 = List(1,2,3)
  val total = sum(example)
}
```

#1 List data type
#2 Data constructor for List
#3 List companion object
#4 Pattern matching example
#5 Variadic function syntax
#6 Creating lists

Let's look first at the definition of the data type, which begins with the keywords sealed `trait`. In general, we introduce a data type with the `trait` keyword. A `trait` is an abstract interface that may optionally contain implementations of some methods. Here we are declaring a `trait`, called `List`, with no methods on it. Adding `sealed` in front means that all implementations of our trait must be declared in this file.[2]

There are two such implementations or *data constructors* of `List` (each introduced with the keyword `case`) declared next, to represent each of the two possible forms a `List` can take—it can be *empty*, denoted by the data constructor `Nil`, or it can be *nonempty* (the data constructor `Cons`, traditionally short for construct), in which case it consists of an initial element, `head`, followed by a `List` (possibly empty) of remaining elements (the `tail`).

Listing 2 The data constructors of List

```
case object Nil extends List[Nothing]
case class Cons[+A](head: A, tail: List[A]) extends List[A]
```

Just as functions can be polymorphic, data types can be as well, and by adding the type parameter `[+A]` after `sealed trait List` and then using that `A` parameter inside of the `Cons` data constructor, we have declared 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 `+` indicates that the type parameter, `A` is covariant—see sidebar "More about variance" for more info).

A data constructor declaration gives us a function to construct that form of the data type (the case object `Nil` lets us write `Nil` to construct an empty `List`, and the case class `Cons` lets us write `Cons(1, Nil)`, `Cons(1, Cons(2, Nil))`, and so on for nonempty lists), but also introduces a pattern that can be used for pattern matching, as in the functions `sum` and `product`.

More about variance

In the declaration, the in `trait List[+A]`, the `+` in front of the type parameter `A` is a variance annotation that signals `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 `+` 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 are not very important for the present discussion and are more of an artifact of how Scala encodes data constructors via subtyping.[3]

---

[2] We could also say `abstract class` here instead of `trait`. Technically, an `abstract class` can contain constructors, in the OO sense, which is what separates it from a `trait`, which cannot contain constructors. This distinction is not really relevant for our purposes right now.

[3] It is certainly possible to write code without using variance annotations at all, and function signatures are sometimes simpler (while type inference often gets worse).

Pattern matching

Let's look in detail at the functions `sum` and `product`, which we place in the object `List`, sometimes called the *companion object* to `List` (see sidebar). Both of these definitions make use of pattern matching.

```
def sum(ints: List[Int]): Int = ints match {
  case Nil => 0
  case Cons(x,xs) => x + sum(xs)
}
def product(ds: List[Double]): Double = ds match {
  case Nil => 1.0
  case Cons(0.0, _) => 0.0
  case Cons(x, xs) => x * product(xs)
}
```

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, `x`, plus the sum of the remaining elements, `xs`.[4] Likewise, the `product` definition 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. Notice these are recursive definitions, which are quite common when writing functions that operate over recursive data types like `List` (which refers to itself recursively in its `Cons` data constructor).

This isn't the most robust test—pattern matching on `0.0` will match only the exact value `0.0`, not `1e-102` or any other value very close to `0`.

Pattern matching works a bit like a fancy `switch` statement that may descend into the structure of the expression it examines and extract subexpressions of that structure (we'll explain this shortly). It is introduced with an expression (the *target* or *scrutinee*), like `ds` followed by the keyword match, and a `{}`-wrapped sequence of cases. Each case in the match consists of a pattern (like `Cons(x,xs)`) to the left of the `=>` and a result (like `x * product(xs)`) to the right of the `=>`. If the target matches the pattern in a case (see below), the result of that case becomes the result of the entire match expression. If multiple patterns match the target, Scala chooses the first matching case.

Companion objects in Scala

We will often declare a *companion object* in addition to our data type and its data constructors. This is just an object with the same name as the data type (in this case List) where we put various convenience functions for creating or working with values of the data type.

If, for instance, we wanted a function `def fill[A](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 for it. Companion objects are more of a convention in Scala.[5] We could have called this module `Foo` if we wanted, but calling it List makes it clear that the module contains functions relevant to working with lists, and also gives us the nice `List(1,2,3)` syntax when we define a variadic apply function (see sidebar "Variadic functions in Scala").

Let's look at a few more examples of pattern matching:

- `List(1,2,3) match { case _ => 42 }` results in `42`. Here we are using a variable pattern, `_`, which matches any expression. We could say `x` or `foo` instead of `_` but we usually use `_` to indicate a variable whose value we ignore in the result of the case.[6]
- `List(1,2,3) match { case Cons(h,t) => h }` results in `1`. Here we are using a data constructor pattern in conjunction with variables to capture or bind a subexpression of the target.

---

[4] We could call `x` and `xs` anything there, but it is a common convention to use `xs`, `ys`, `as`, `bs` as variable names for a sequence of some sort, and `x`, `y`, `z`, `a`, or `b` as the name for a single element of a sequence. Another common naming convention is `h` for the first element of a list (the "head" of the list), `t` for the remaining elements (the "tail"), and `l` for an entire list.

[5] There is some special support for them in the language that isn't really relevant for our purposes.

[6] The `_` variable pattern is treated somewhat specially in that it may be mentioned multiple times in the pattern to ignore multiple parts of the target.

- `List(1,2,3) match { case Cons(_,t) => t }` results in `List(2,3)`.
- `List(1,2,3) match { case Nil => 42 }` results in a `MatchError` at runtime. A `MatchError` indicates that none of the cases in a match expression matched the target.

What determines if a pattern matches an expression? A pattern may contain *literals*, like 0.0 or `"hi"`, *variables* like `x` and `xs` which match anything, indicated by an identifier starting with a lowercase letter or underscore, and data constructors like `Cons(x,xs)` or `Nil`, which match only values of the corresponding form (`Nil` as a pattern matches only the value `Nil`, and `Cons(h,t)` or `Cons(x,xs)` as a pattern only match `Cons` values). These components of a pattern may be nested arbitrarily—`Cons(x1, Cons(x2, Nil))` and `Cons(y1, Cons(y2, Cons(y3, _)))` are valid patterns. A pattern matches the target if there exists an assignment of variables in the pattern to subexpressions of the target that make it *structurally equivalent* to the target. The result expression for a matching case will then have access to these variable assignments in its local scope.

EXERCISE 1: What will the result of the following match expression be?

```
val x = List(1,2,3,4,5) match {
  case Cons(x, Cons(2, Cons(4, _))) => x
  case Nil => 42
  case Cons(x, Cons(y, Cons(3, Cons(4, _)))) => x + y
  case Cons(h, t) => h + sum(t)
  case _ => 101
}
```

You are strongly encouraged to try experimenting with pattern matching in the REPL to get a sense for how it behaves.

Variadic functions in Scala

The function `List.apply` in the listing above is a variadic function, meaning it accepts zero or more arguments of type `A`. For data types, it is a common idiom to have a variadic `apply` method in the companion object to conveniently construct instances of the data type. By calling this function `apply` and placing it in the companion object, we can invoke it with syntax like `List(1,2,3,4)` or `List("hi","bye")`, with as many values as we want separated by commas (we sometimes call this the list *literal* or just *literal syntax*).

Variadic functions are just providing a little syntax sugar for creating and passing a `Seq` of elements explicitly. `Seq` is the interface in Scala implemented by sequence-like data structures like lists, queues, vectors, and so forth. Inside apply, as will be bound to a `Seq[A]` (documentation link), which has the function's `head` (returns the first element) and `tail` (returns all elements but the first).

We can convert a `Seq[A]`, `x`, back into something that can be passed to a variadic function using the syntax `x: _*`, where `x` can be any expression, for instance: `List(x: _*)` or even `List(List(1,2,3):_*)`.

Summary

In this article, we covered a number of important functional programming concepts related to Scala. We introduced algebraic data types and pattern matching.

Here are some other Manning titles you might be interested in:

The Real-World Functional Programming
Tomas Petricek with Jon Skeet

Scala in Action
Nilanjan Raychaudhuri

Play for Scala
Peter Hilton, Erik Bakker, and Francisco Canedo

Last updated: November 3, 2012