Functional Programming Style
Drew McDermott
drew.mcdermott@yale.edu — 2015-09-30 & 10-02
Revised 2015-10-15, by addition of paragraph on type ascription in case clauses at the end of the section “Variable Number of Arguments.”

Having learned recursion, you may think that’s all there is to functional programming. For better or for worse, there’s a lot more. We will only scratch the surface in this course. You may get hooked on functional programming (as many people do) and want to dig deeper.

The basic idea we will look at is this: Given some input data structures, and a desired output data structure, think of a way to transform the input as a whole into the output, one step at a time. The archtypical example of the species is map. In Scala, it is a method member of every collection class. If \( x l : \text{List[Int]} \), then \( x l . m a p ( f ) \) produces a \( \text{List[B]} \) if \( f : \text{Int} \Rightarrow B \). Example: \( x l . m a p ( n \Rightarrow "a" \ast n ) \)

Another useful method is filter. \( l . f i l t e r ( p ) \) is a collection of all the elements of \( l \) that satisfy predicate \( p \), e.g., \( \text{strings.filter(s \Rightarrow s.length} == k) \).

Exercise: Suppose we have a list of lists of strings. We want to return a list containing all the shortest strings from each sublist. E.g., if \( y l = \text{List(List("a", "bc", "c"), List("abcdefg", "abcd"), List())} \), we want the result \( \text{List("a", "c", "abcd")} \). (I can’t imagine why.) But let’s say we want to avoid val for obscure pedagogical purposes, and also want to avoid finding the shortest string multiple times.

The solution is given below. In case you weren’t in class when we discussed it, try to solve it yourself before looking at the solution Doing so will require two important new tools (zip and flatMap), and others not so important.

First, some further remarks on collections and operations on them:

- map, filter, and other operations are applicable to any collection type, even Option[T]. If \( x : \text{Option[T]} \), then \( x . m a p ( f ) \) is None if \( x == \text{None} \), Some(\( f(y) \)) if \( x == \text{Some}(y) \).

- Strings are not collections, but they behave like them in almost every way. For example,

  \( \text{"abc".map(c \Rightarrow List(c))} \rightarrow \text{Vector(List(a), List(b), List(c))} \)

  (where \( e \rightarrow v \) means “\( e \) evaluates to \( v \)”.) The output in this case can’t be a string, so Scala chooses to make it a Vector. If you don’t like that, you can convert the vector to a list, or convert the string:
"abc".toList.map(c => List(c)) → List(List(a), List(b), List(c)

But the documentation for Collections (and many other classes) have many members of the form toF, where F is another form, like toChar in class Int or toArray in class String, which converts a string to an array of characters.¹

Flattening and flatMap

If you have a list of lists \( ll = \text{List}(l_1, \ldots, l_k) \), and you want to convert it to the list \( l_1++l_2++\ldots++l_k \), where “++” is the operation that concatenates two things, there is a built-in operation \( ll.\text{flatten} \) that performs exactly that transformation. (It removes a layer of parentheses, making the structure 1 level “shallower,” hence the name.)

If you’re generating such a list, you can flatten as you go by using flatMap. For example, given a list of strings, suppose you want to find all the characters that occur in any string. A first step might be

\[
\text{listOfStrings.} \text{flatMap(s => s.toList)}
\]

Now that we have flatMap, here’s one way to solve our problem of producing a list of all strings that are of minimal length in their sublist:

\[
\text{yl.} \text{flatMap(sublist => { if (sublist.length == 0) List() else { val shortestLen} \text{sublist.map(_.length).min \text{sublist.filter(s => s.length == shortestLen) } })}
\]

Unfortunately, this violates our constraint on using val, and just substituting sublist.map(_.length).min for shortestLen in the last line will cause the program to recalculate the minimum for every element of the list. We

¹For hairy reasons, it’s actually in a class scala.collection.immutable.StringOps, along with many other operations that Java doesn’t provide or has a different name for. The class String is mysteriously missing from the Scala-library documentation, because it’s actually exactly the same as the class String from Java (whose full name is java.lang.String). All the goodies are in StringOps. For example, because StringOps supplies an apply method, you can just write \( s(i) \) instead of \( s.getCharAt(i) \), although the latter works. The existence and usefulness of StringOps is just something you have to know, unfortunately.
need another way to save intermediate values and bring them together with other lists. That way will involve the \texttt{zip} method. But first, some words about \texttt{for} loops in Scala.

\textbf{for Expressions}

The \texttt{for} statement in Scala is very different from the silly C-style \texttt{for}, which is just an anemic, misleading abbreviation for \texttt{while}. It has two forms, one imperative and one functional. The imperative form is

\begin{verbatim}
for (v1 <- g1,
    v2 <- g2,
    ...
    v_k <- g_k)
  C
\end{verbatim}

Suppose for simplicity all the $g_i$ are lists. This iterates through each element of $g_1$, binding $v_1$ successively to each element. For each value of $v_1$, it iterates through $g_2$, and so forth, so that it evaluates $C$ with respect to all combinations of values generated. In other words, it’s roughly equivalent to

\begin{verbatim}
for (v1 <- g1)
  for (v2 <- g2)
    ...
    for (v_k <- g_k)
      C
\end{verbatim}

The form is imperative because if $C$ doesn’t have side effects, the exercise is pointless or ugly.

But $g_i$ doesn’t have to be a list. It can be any kind of collection, including a \texttt{range}, a finite arithmetic progression, such as “0 to 9” or “10 until 0 by -2”. (The difference between \texttt{to} and \texttt{until} is that the former includes the second number and the second does not. So “for (i <= 0 until a.length...)” is like the classical C-style loop “for (int i=0; i < a.length; i++)...”.)

But never mind the imperative case; we’re trying \textit{not} to write loops! The other form of \texttt{for} is called a \textit{comprehension}; it looks like this:

\begin{verbatim}
for {
  v1 <- g1
  v2 <- g2
    ...
  v_k <- g_k
} yield E
\end{verbatim}
which generates the combinations in exactly the same way, but evaluates E for each combination, and collects all the results into a new collection whose elements have the same type as E. (Why braces instead of parens? Because the Scala style guide suggests it. I don’t think it matters, really.)

*Important:* The result collection has the type of \(g_1\), so if that’s not right you have to either transform \(g_1\) or transform the result. For instance, if \(g_1\) is a Vector[Int], and E is of type String, then the for-comprehension is of type Vector[String]. If the system can’t figure out how to pull this off, the type is only predictable from experience. In particular, if \(g_1\) is a range, it’s unlikely the output will be a range. Best to transform \(g_1\) to the desired type.

In other words, if \(g_1\) is a range, \(E\) is a String, and you want a List[String] in the end, write

```scala
for {
  v_1 <- g_1.toList
  ...
  v_k <- g_k
} yield E
```

Collections (and other classes) usually have a toT method for any reasonable type \(T\).

How would you produce the same result with map and flatMap? The transformation is quite simple. Every <- becomes a \(\lambda\)-expression, which is the argument to a flatMap, except for \(v_k\), where it is a map:

\[
g_1.flatMap(v_1 =>
g_2.flatMap(v_2 =>
  ...\n  g_k.map(v_k => E)))
\]

One problem with Scala’s for is that there’s no way to terminate the loop early. There’s a standard imperative idiom using a while loop to look for something:

```scala
var found: Boolean = false
var i: Int = 0
while (!found && i < v.length) {
  ...
  if (...) {

2If you really really want a function that finds a way to express the result as a range and throws an exception if it can’t, you’re going to have to write it.
The loop terminates when it runs out of things to look at or `found` becomes true. At the end the value of `found` tells you whether the search was successful.

Unfortunately, in Scala you can’t use `for`, imperatively or otherwise, to write this search. The closest you can come is

```scala
var found : Boolean = false
for (i <- 0 until v.length if !found) {
  ...
  if (...) {
    found = true
  } else {
    ...
  }
}
```

The difference is this: If `v.length` is 10,000, and `found` becomes true when `i==5`, the `for` while spin its wheels through the last 9994 iterations, skipping the body of the loop, but running through all the values of `i`.³

But cheer up! There are ways to do searches functionally. Here are two:

1. If `c` is a collection, `c.find(pred)` returns the first element of `c` satisfying `pred`. Oops, there might not be one; so it actually returns an `Option[t]`, where `t` is the type of the elements of `c`.

2. If `c` is a collection, `c.indexWhere(pred, i)` returns the index (i.e., the position) of the first element satisfying `pred` on or after position `i` (default is 0.). If there is no such element, the value is \(-1\).

³Of course, a smart enough compiler could optimize out this waste, but I doubt it’s that smart. But while I’m whispering in this footnote, you might want to check out the object `scala.util.control.Breaks`, which provides a way to break out of a `for` or anything else, unfortunately by throwing an exception, which seems perverse. I generally disapprove of `break`, and there are better ways of doing searches than using `while`; read on.
Obviously, the latter is more likely to be useful in collections that allow random access to elements by index, such as arrays and vectors.

**zip and Partial Functions**

Suppose we have two lists $l_1: \text{List}[E_1] = \text{List}(l_{10}, l_{11}, \ldots, l_{1,n-1})$ and $l_2: \text{List}[E_2] = \text{List}(l_{20}, l_{21}, \ldots, l_{2,n-1})$. We have a function $ff: (E_1, E_2) \Rightarrow R$, and we’d like to apply it successively to $l_{10}$ and $l_{20}$, $l_{11}$ and $l_{21}$, and so on through $l_{1,n-1}$ and $l_{2,n-1}$.

The only way to do that in Scala is to use `zip` and some other machinery.

11 `zip` 12 produces the list of pairs

$$\text{List}((l_{10}, l_{20}), (l_{11}, l_{21}), \ldots, (l_{1,n-1}, l_{2,n-1}))$$

To apply $ff$ we have to deconstruct what we’ve constructed:

$$(l_1 \text{ zip } l_2).\text{map}(p \mapsto p \text{ match } \{
\text{case } (e_1, e_2) \Rightarrow ff(e_1, e_2)
\})$$

This can be abbreviated by using what’s called an *anonymous partial function*:

$$(l_1 \text{ zip } l_2).\text{map}(
\{
\text{case } (e_1, e_2) \Rightarrow ff(e_1, e_2)
\})$$

In other words, the `case` construct, wrapped in braces, becomes a function of one argument that does pattern matching on that argument.\(^4\)

For instance, suppose you had a $\text{List}[\text{List}[$String$]]$, once again, and you’d like to find the first string in each list with the same length as the list itself, or the empty string if there is no such string, without using `val`. Here’s an example:

```scala
val zl: \text{List}[$\text{List}[$"abc", "bcd", "c"], $\text{List}[$"abcdefg", "abcd"], $\text{List}()$, $\text{List}[$"a"])
```

\(^4\)For further discussion of some of the things you can do with partial functions, see [http://blog.bruchez.name/2011/10/scala-partial-functions-without-phd.html](http://blog.bruchez.name/2011/10/scala-partial-functions-without-phd.html)
For which the desired result is \texttt{List("abc", ",", ",", "a")}, because "abc" is the first string of length 3 in its sublist, the second sublist has no string of length 2, the third has no string of length 0, and the fourth has exactly one string of length 1, namely, "a".

We can produce a list of the lengths of the component lists easily enough, as \texttt{zl.map(_.length)}. We then bring this together with the original list using \texttt{zip}: \texttt{zl zip (zl.map(_.length))}. We know we want to produce a list with the same length as \texttt{zl}, but with string elements. That suggests a map:

\texttt{(zl zip (zl.map(_.length))).map(...)}

The dots need to be replaced by a function definition. It has one parameter, the pair we’re getting from \texttt{zip}. We’ll deconstruct it using an anonymous partial function:

\texttt{(zl zip (zl.map(_.length))).map({
  case (sublist, len) => ...
})}

Whew! Now what we want is to find the first string in \texttt{sublist} that has the length \texttt{len}. We’re almost there if we use \texttt{find}:

\texttt{(zl zip (zl.map(_.length))).map({
  case (sublist, len) =>
    sublist.find(s => s.length == len)
}),}

the only problem being that \texttt{find} returns an \texttt{Option[String]}. No problem; we can use the \texttt{getOrElse} method of class \texttt{Option}, which allows us to convert \texttt{Option[A]} to \texttt{A}, if we supply a default value for the \texttt{None} case. The final piece of code is:

\texttt{(zl zip (zl.map(_.length))).map({
  case (sublist, len) =>
    sublist.find(s => s.length == len).getOrElse("")
})}

\textbf{Variable Number of Arguments And Other Issues}

\footnote{This will print out as \texttt{List(abc, , , a)}, because Scala uses the same \texttt{toString} function for printing everything, and obviously you want to be able to print strings without quotes. Perhaps there should be a \texttt{toDebugString} method that by default calls \texttt{toString} but is overridden in the \texttt{String} class to include the quotes.}
In every language this issue sooner or later comes up. How do you define and use a function with a variable number of arguments? In Scala, you do it by putting an asterisk after the type of the last parameter:

```scala
def fcnWithVarargs(x: Int, y: Int, lastArg: String*) ...
```
defines `fcnWithVarargs` to take a variable number of `String` arguments after the first two, which must be `Int`s. The type of `lastArg` is `Seq[String]`. `Seq` is a trait, so that doesn’t tell you what the run-time type is, but you can pretty much assume that anything you can do to a `List` you can do to a `Seq`.

But can you write a function that behaves like `append` in Lisp or Racket? Its declaration would look like:

```scala
def append[A](lists: List[A]*): List[A] = {
  lists.foldRight(Nil)((++)
}
```

As explained in the PS1 solutions, if `ll = List(l0, l1, ..., ln−1)`, then

```scala
ll.foldRight(Nil)((++) =
  (l0 ++ (l1 ++ ... (ln−2 ++ ln−1) ...))
```

which is exactly what we want.

Because “++” is associative, we could also have used `foldLeft`:

```scala
ll.foldLeft(Nil)((++) =
  ((...(l0 ++ l1) ++ ... ++ ln−2) ++ ln−1)
```

However, doing it this way requires building and copying a longer and longer list each time, whereas `foldRight` copies each list at most once.

Finally, suppose you already have a list of lists, and you want to apply `append` to that list and bind the parameter `lists` to it. (In other words, you want to use `append` instead of `flatten`.) You have to do something a little ugly: add a bogus `type ascription` to the argument when you call it:

```scala
append(listOfLists : _*)
```

---

6It bubbled up CS470/570 when Prof. Angluin asked what the Scala equivalent of `(apply append l)` in Scheme (or Racket) was.
You can write “: type” after any expression to “help the compiler” infer the correct type for that expression. This is yet another use of underscore in Scala to mean “whatever it would be useful for underscore to mean.” We’re not really telling the system anything about the type of `listOfLists` except that we want it to grab that asterisk that normally goes to the argument list of `append`.

Here’s another example: Suppose we define a function that subtracts from its first argument the sum of its remaining arguments:

```scala
def takeFrom(a: Int, l: Int*): Int = a - l.foldLeft(0)(+_
```

So `takeFrom(100, 1, 2, 3)` would evaluate to 94. Then we can subtract from the head of a list of ints the sum of the elements of its tail by evaluating `takeFrom(ints.head, ints.tail : _*)`.\(^8\)

What about the reverse transformation, during pattern matching? Suppose there’s a family of case classes, one of which take an indefinite number of arguments; suppose that one’s name is `mult`. In a match, can we write

```scala
exp match {
  case ... => ...
  case mult(l: _) => ... // Illegal!
}
```

Surprisingly, no. You have to write

```scala
exp match {
  case ... => ...
  case mult(l@ _) => ...
}
```

which differs by one character: the substitution of “@” for “:”. It’s a complete mystery why.\(^9\)

---

7 For example, sometimes the compiler needs to be told what `Nil` is an empty list of. The way you tell it is by writing “`Nil: List[?]`.”

8 It would be cool if we could just write `takeFrom(ints: _)`, but that’s too far toward Racket.

9 If I had to guess at an explanation, here’s my guess: The left side of a case clause is a variable-binding specification; every variable mentioned is freshly bound. (There is a way around this, but let’s ignore it.) An ordinary type ascription is interpreted as a `declaration` of that variable once it’s successfully matched, i.e., after the match goes through. That type must be consistent with the type inferred by the matcher’s mechanism (called `unapply`), but perhaps less specific in a useful way. Once the match succeeds, the ascribed type is the type used when the right-hand side of the case clause is evaluated. The at-sign mechanism applies before that match goes through. Fortunately, I haven’t done the obvious experiments to test this theory.
A Few Useful “Laws”

When writing and reading functional code, you need a few rules to cling to, such as

1. \( l . \text{map}(f) \) has the same number of elements as \( l \)
2. \( l . \text{filter}(p) \) has the same type of elements as \( l \)
3. \( \text{for } \{ \; v < - g \; \ldots \} \text{ yield } E \) is a collection of the type of \( g \), with elements the type of \( E \).

Solution to Problem

You have enough tools to solve the problem of finding a list of all strings that are the shortest in some component list of the \( \text{List[\text{List[\text{String}]!!.}}} \) we started with, without using \text{val}.

So go solve it. Then come back here.

Don’t Peek!

Okay, here is the solution to the problem: Given a list \( yl \) of type \( \text{List[\text{List[\text{String}]!!.}}} \), produce a list of strings that are the shortest string in one of the component lists. E.g., if

\[
yl == \text{List(List(\text{"ab"}, \text{"abc"}, \text{"bc"}), \text{List()}, \text{List(\text{"abc"}, \text{"abcd"})})},
\]

the output we want is

\[
\text{List(\text{"ab"}, \text{"bc"}, \text{"abc"})}
\]

The string "abc" is in the list because it is the shortest in the third component list, even though it is not the shortest in the first. Both "ab" and "bc" are in the result because each is of length 2, and that’s as short as the strings get in that list.

The computation naturally breaks down into two pieces, pre-zip and post-zip. This is the zip:

\[
yl . \text{zip}(yl . \text{map}(\{\}
    \text{ case Nil } => 0
    \text{ case sublist } => \text{sublist.map(_.length).min}
    \}))
\]

Now we apply \text{flatMap} to the pairs:
You can try using for to make this more perspicuous, but it won’t help much. What it needs is to be broken up into smaller expressions with val, which in practice is exactly what we’d do. Even so, in realistic cases, it is still often necessary to use zip.