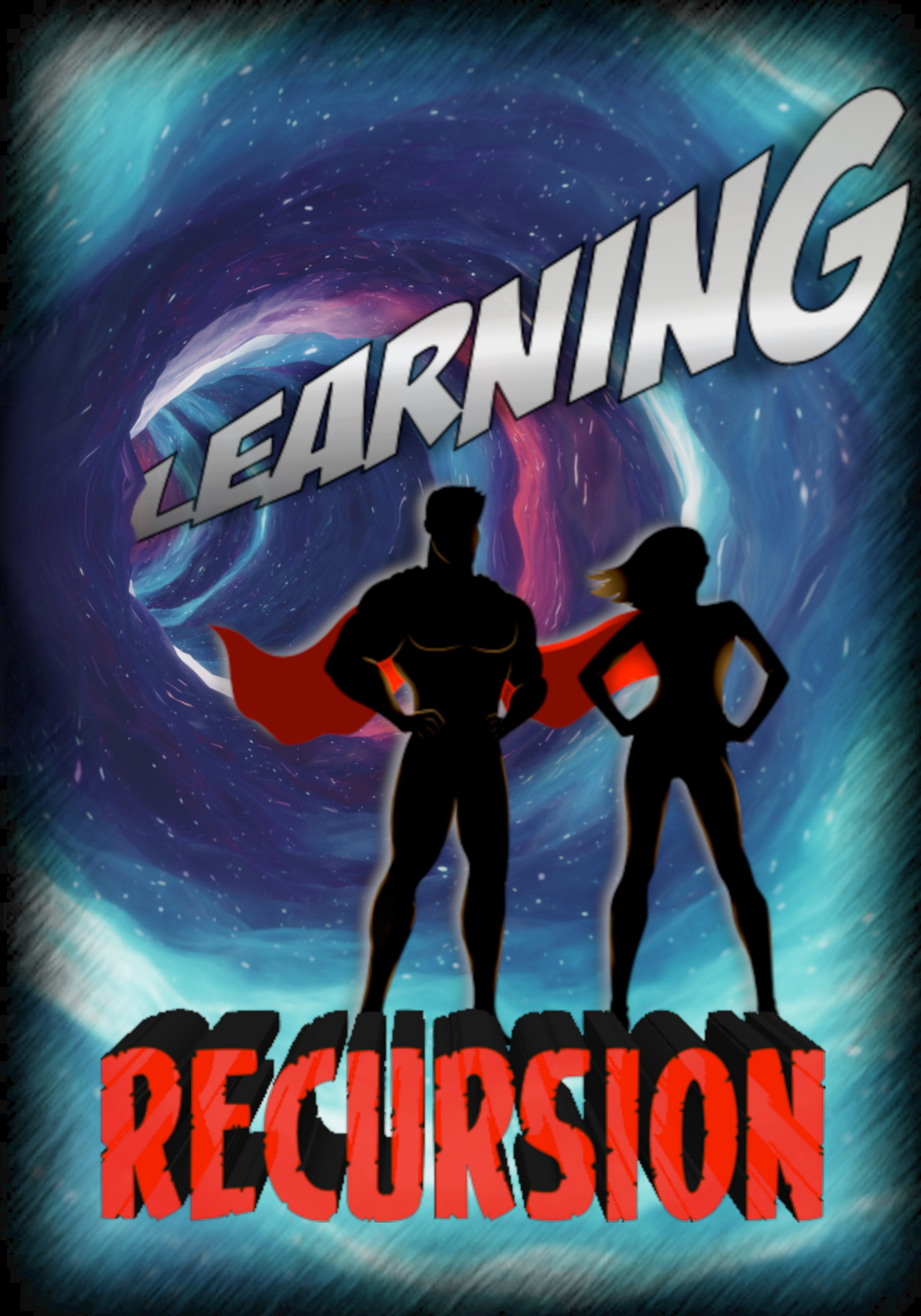

LEARNING
RECURSION

Learning Recursion

Alvin Alexander

Copyright

Learning Recursion

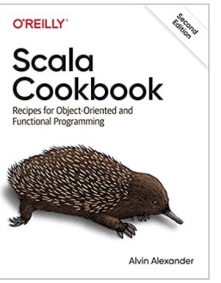
Version 0.1, published January 23, 2023

The "superheroes" on the front cover are "Illustration 105920523 © Bonezboyz | Dreamstime.com."
The wormhole on the front cover is "Illustration 149478810 / 3d © Rostislav Zatonskiy | Dreamstime.com."

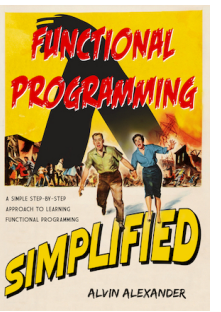
[1] https://alvinalexander.com

Other books

Other books by Alvin Alexander:

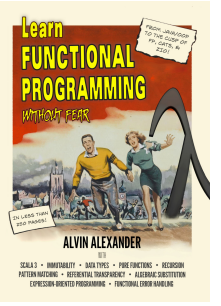


Scala Cookbook, 2nd Edition (Amazon.com)[2]



Functional Programming, Simplified
("The Big FP Book," alvinalexander.com)[3]

[2] https://amzn.to/3du1pMR
[3] https://alvinalexander.com/scala/functional-programming-simplified-book



Learn Functional Programming Without Fear
("The Little FP Book," alvinalexander.com)[4]

Learn Scala 3 The Fast Way![5]

[4] https://alvinalexander.com//scala/learn-functional-programming-book
[5] https://alvinalexander.com/scala/learn-scala-3-the-fast-way-book

Contents

1

Welcome

Welcome! This is a free booklet on *recursion*, also known as recursive programming.

The way this booklet came into being is that in January, 2023, I started updating my book, Functional Programming, Simplified[1], to use Scala 3 and other modern functional programming (FP) libraries like ZIO[2] and Cats Effect[3]. When I got to these chapters on recursion, I realized that they were almost 100 pages long all by themselves, so I decided to pull them out into this booklet for any programmers who are interested in recursion, but aren't interested in all the other FP content in that book.

Who this book is for

In the beginning of almost all of my books I include a "Who this book is for" chapter, but in this case that's apparent: It's for anyone who is interested in learning about *recursion*. The examples are written using Scala 3, but I don't do anything fancy, so the code and techniques should translate well to other languages.

[1] https://alvinalexander.gumroad.com/l/fpsimplified
[2] https://zio.dev
[3] https://typelevel.org/cats-effect

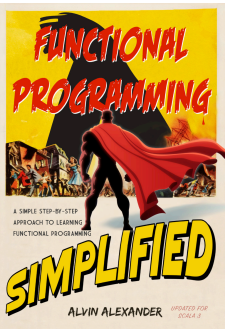
Goals

I also usually include a "Goals" chapter in each book, but again, that's pretty apparent for this book: The goal is to share a collection of lessons about programming using recursion.

Recursive programming just involves functions that call themselves, so in this booklet I'll show examples of how to write these types of functions.

Functional Programming, Simplified: Updated for Scala 3

As I mentioned, at the time of this writing in January, 2023, I'm currently updating my book, *Functional Programming, Simplified*, to work with Scala 3, *ZIO*, and *Cats Effect*. If you're interested in reading that book as I update it, you can find those PDF releases here:



Functional Programming, Simplified: Updated For Scala 3[4]

And now, let's jump into the recursion lessons!

All the best,
Al

[4] https://alvinalexander.gumroad.com/l/fpsimplified

2

Recursion: Introduction

You're about to jump into a series of lessons on recursive programming. Please note that some of these lessons may be overkill for some people. Basically what I do in the following chapters is introduce recursion in several different ways — for example, using code, using a conversation between two developers, and using images — so if one of those ways works for you, great! Whenever you understand recursion, just stop where you are and don't worry about reading the remaining lessons.

> The one exception I'll add to that statement is that you may still want to review the lessons on stacks, stack frames, and tail recursion, because they add more knowledge on top of the "basic recursion" technique.

3

Recursion: Motivation

"To iterate is human, to recurse divine."

L. Peter Deutsch

What is recursion?

Before getting into the motivation to use recursion, a great question is, "What is recursion?"

Simply stated, a *recursive function* is a function that calls itself. That's it.

Why do I need to write recursive functions?

In terms of motivation, the next question that usually comes up right about now is, "Why do I need to write recursive functions? Why can't I use `for` loops to iterate over lists?"

The short answer is that algorithms that use `for` loops require the use of `var` fields, and as I mention in my book, *Functional Programming, Simplified*, we never us `var` fields or mutable data structures in functional programming (FP).

(Read on for the longer answer.)

> Please recall that these lessons come from my FP book. That's why this answer comes from an FP perspective.

If you had `var` fields

Of course if you *could* use mutable variables in your programming language, you might write a "sum the integers in a list" algorithm like this:

```
def sum(xs: List[Int]): Int =
    var sum = 0
    for x <- xs do
        sum += x
    sum
```

This algorithm uses a `var` field named `sum` and a `for` loop to iterate through every element in the list that's passed into the function as the input parameter `xs`. In Scala this is how you use this function to print the sum of a small list of integers:

```
val ints = List(1, 2, 3)
println(sum(ints))      // the result is 6
```

From an imperative programming standpoint, there's nothing wrong with this code. I wrote imperative code like this in Java for more than fifteen years. But from a functional programmer's point of view, there are several problems with this code.

Problem 1: We can only keep so much in our brains

One problem is that reading a lot of custom `for` loops dulls your brain.

As an OOP/imperative programmer I never noticed it, but if you *think*

about the way you thought when you read that function, one of the first things you thought is, "Hmm, here's a `var` field named `sum`, so Al is probably going to modify that field in the rest of the algorithm." Next, you thought, "Okay, here's a `for` loop … he's looping over `xs` … ah, yes, he's using `+=`, so this really is a 'sum' loop, so that variable name makes sense."

Once you learn FP — or even if you just learn the functional methods available on the Scala collections classes — you realize *that's an awful lot of thinking* about a little-bitty custom `for` loop.

If you're like me a few years ago, you may be thinking that what I just wrote is overkill. You might think, "I look at mutable variables and custom `for` loops all the time; what's the big deal?"

The big deal is occasionally known as Miller's Law, or more commonly, The Magical Number Seven, Plus or Minus Two[1]. Simply stated, per that Wikipedia page, "the number of objects an average human can hold in short-term memory is 7 ± 2."

Therefore, one can argue that when you have to read a custom `for` loop like that one, you're filling up those short-term memory slots with mostly useless information. Put differently, since we can only keep *just so much* information in our brains at one time:

- Any time we can keep less information there, it's a win, and
- Boilerplate `for` loop code is a waste of our brain's RAM (and CPU)

Maybe this seems like a small, subtle win at the moment, but speaking from my own experience, anything I can do to keep my brain's RAM free for important things is a win.

[1] https://en.wikipedia.org/wiki/The_Magical_Number_Seven,_Plus_or_Minus_Two

Problem #2: This isn't algebra

From an FP perspective, another problem is that this code doesn't look or feel like algebra. I discussed how important algebra is to FPers in the "Functional Programming is Like Algebra" lesson, so I won't repeat that discussion here, other than to say that FPers like to see their code as algebra (or blueprints).

> *FPer* is short for "functional programmer."

Problem #3: There are no `var` *fields in FP*

Of course from our perspective as functional programmers, the *huge* problem with this code is that it requires a `var` field, and Scala/FP developers (and mathematicians) never use those. What I found in my own research into FP is that `var` fields are a crutch, and the best thing you can do to expedite your FP education is to completely forget that they exist.

My experience was that once you let go of `var` fields and `for` loops, you can discover a different way to solve iterative problems.

What to do?

Because we can't use `var` fields, we need to look at a different tool to solve problems like this. That tool is *recursion*.

If you're like me, at first you'll NEED to write recursive functions because *that's all you're allowed to do in FP*, but after a while you'll WANT to write recursive functions.

4

Recursion Background: Let's Look at Lists

> *"In computer science, a linked list is a linear collection of data elements, called nodes, each pointing to the next node by means of a pointer."*
>
> Wikipedia's Linked List entry[a]

[a] https://en.wikipedia.org/wiki/Linked_list

Visualizing lists

Because the `List` data structure — and the *head* and *tail* components of a `List` — are so important to recursion, it helps to visualize what a list and its head and tail components look like. Figure 4.1 shows one way to visualize a `List`.

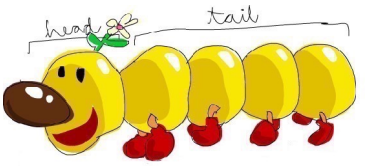


Figure 4.1: One way to visualize the head and tail components of a list.

This creative imagery comes from the online version of "Learn You a

Haskell for Great Good"[1], and it does a terrific job of imprinting the concept of head and tail components of a list into your brain. As shown, the "head" component is simply the first element in the list, and the "tail" is everything else — the rest of the list.

A slightly more technical way to visualize the head and tail of a list is shown in Figure 4.2.

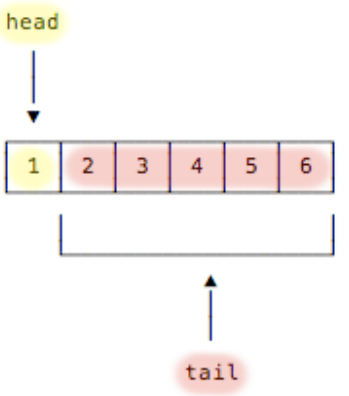


Figure 4.2: A slightly more technical way to visualize a list.

An even more accurate way to show this is with a `Nil` value at the end of the `List`, as shown in Figure 4.3, because that's what a `List` really looks like in Scala.

Linked lists and "cons" cells

To be clear, the `List` that I'm talking about is a *linked list* — the Scala immutable `List`[2], which is the default list you get if you type `List` in your IDE or the REPL:

```
scala> val xs = List(1, 2, 3)
val xs: List[Int] = List(1, 2, 3)
```

[1] http://learnyouahaskell.com/starting-out
[2] https://www.scala-lang.org/api/current/scala/collection/immutable/List.html

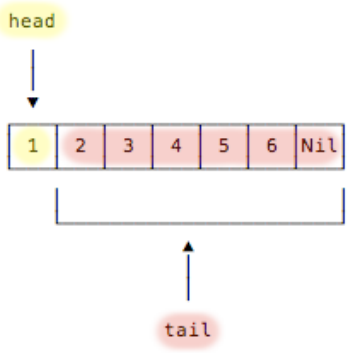


Figure 4.3: A more accurate way to visualize a list.

As shown in Figure 4.4, this `List` is a series of *cells*, where each cell contains two things: (a) a value, and (b) a pointer to the next cell.

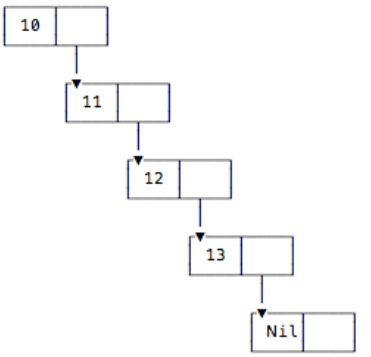


Figure 4.4: An accurate depiction of a linked list, as a series of cells.

What's extremely important for our needs is that the last cell in a linked list contains the `Nil` value. This value being in the last cell is important because it's how your recursive Scala code will know when it has reached the end of a `List`.

Building on that previous image, Figure 4.5 highlights the head element of a list, and Figure 4.6 highlights the tail elements. Just like the Haskell programming language — and Lisp before it — the default Scala `List` works with these head and tail components, and I'll use them extensively in the examples that follow.

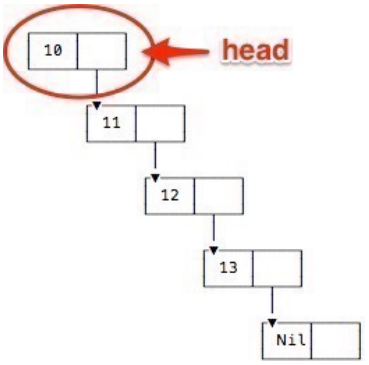


Figure 4.5: The head element of a list.

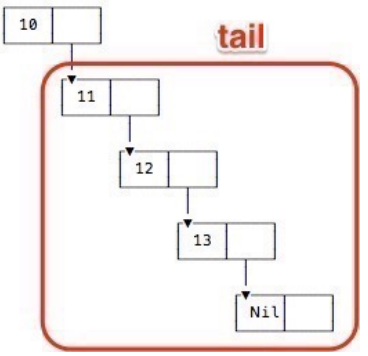


Figure 4.6: The tail elements of a list.

> For historical reasons these cells are known as *cons cells*. That name comes from Lisp, and if you like history, you can read more about it on Wikipedia[a].
>
> [a] https://en.wikipedia.org/wiki/Cons

Note 1: The empty `List`

A `List` with no elements in it is an *empty list*. An empty `List` contains only one cell, and that cell is the `Nil` element, as shown in Figure 4.7.

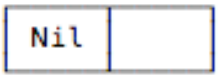


Figure 4.7: A list with no elements contains only one cell, which is the `Nil` *element.*

You can create an empty `List` in Scala in a few different ways:

```
scala> val empty = List()
empty: List[Nothing] = List()

scala> val empty = Nil
empty: scala.collection.immutable.Nil.type = List()
```

Because I didn't give those lists a data type (like `Int`), the results look a little unusual, and as a practical matter you probably won't do this in the real world. What you *will* do is specify a data type when you create your lists, like this:

```
scala> val empty1: List[Int] = List()
empty: List[Int] = List()

scala> val empty2: List[Int] = Nil
empty: List[Int] = List()
```

```
scala> empty1 == empty2
res0: Boolean = true
```

Note 2: Several ways to create `Lists`

There are several ways to create *non-empty* `Lists` in Scala, but for the most part I'll use two approaches. First, here's the most-common approach:

```
val list = List(1, 2, 3)
```

Second, this is an approach you may not have seen before:

```
val list = 1 :: 2 :: 3 :: Nil
```

These two techniques result in the exact same `List[Int]`, which you can see in the REPL:

```
scala> val list1 = List(1, 2, 3)
list: List[Int] = List(1, 2, 3)

scala> val list2 = 1 :: 2 :: 3 :: Nil
list: List[Int] = List(1, 2, 3)

scala> list1 == list2
res1: Boolean = true
```

The second approach is known as using "cons cells." As you can see, it's a very literal approach to creating a `List`, where you specify each element in the `List`, including the `Nil` element, which must be in the last position. The elements you specify are literally the values in each cons cell. Note that if you forget the `Nil` element at the end, the Scala compiler will give you an error message:

```
scala> val list = 1 :: 2 :: 3
-- [E008] Not Found Error:
```

```
-------------------------
1 |val list = 1 :: 2 :: 3
  |                  ^^^^
  |                  value :: is not a member of Int
1 error found
```

I emphasize this because it's important — very important — to know that the last element in a `List` MUST be the `Nil` element. (The `Nil` element is to a `List` as a caboose is to a train.) We're going to take advantage of this knowledge as we write our first recursive function.

5

Recursion: How to Write a 'sum' Function

With all of the images of the previous lesson firmly ingrained in your brain, let's write a `sum` function using recursion!

Sketching the `sum` function signature

Given a `List` of integers, such as this one:

```
val list = List(1, 2, 3, 4)
```

let's start tackling the problem in the usual way, by thinking, "Sketch the function signature first."

> NOTE: I write all functions in Functional Programming, Simplified[a] by first sketching the function signature, and then implementing the body of the function. This is the way we think when solving problems and writing functions in FP.
>
> [a] https://alvinalexander.com/scala/functional-programming-simplified-book

What do we know about the `sum` function we want to write? Well, right off the bat we know a couple of things:

- It will take a list of integers as input
- Because it returns a sum of those integers, the function will return a single value, an `Int`

Armed with those two pieces of information, I sketch the signature for a `sum` function like this:

```
def sum(list: List[Int]): Int = ???
```

> NOTE: For the purposes of this example, I'm assuming that the integer values will be small, and the list size will also be small. That way we don't have to worry about all of the `Int`s adding up to a `Long` or `BigInteger`.

The `sum` function body

At this point an FPer will think of a "sum" algorithm as follows:

1. If the `sum` function is given an empty list of integers, it should return `0`. This is because "the sum of nothing" is zero.
2. Otherwise, if the list is *not* empty, the result of the function is the combination of (a) the value of its head element (`1`, in our example), and (b) the sum of the remaining elements in the list (`2,3,4`).

A slight restatement of that second sentence is:

> "The sum of a list of integers is the sum of the *head* element, plus the sum of the *tail* elements."

Now that we have a little idea of how to *think* about the problem recursively, let's see how to implement those sentences in Scala code.

> TIP: Thinking about a `List` in terms of its head and tail elements is a standard way of thinking when writing recursive functions.

Implementing the first sentence in code

The first sentence above states:

> If the `sum` function is given an empty list of integers, it should return `0`.

Recursive Scala functions are commonly implemented using `match` expressions. Using (a) that information and (b) remembering that an empty list contains ONLY the `Nil` element, you can start writing the body of the `sum` function like this:

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    // more code here ...
```

This is a Scala way of saying, "If the `List` is empty, yield `0`." If you're comfortable with `match` expressions and the `List` class, I think you'll agree that this makes sense.

> TIP: When writing a recursive function, write the *end condition* first, just like this. This will give you comfort that the recursive calls will stop when this condition is reached.

Note 1: Using `return`

If you prefer using `return` statements at this point in your programming career, for just this moment you can write that code like this:

```
def sum(list: List[Int]): Int = list match
    case Nil => return 0
    // more code here ...
```

However, because pure functions don't "return" a value as much as they *evaluate to a result*, this is the moment in life where I recommend dropping the `return` keyword from your vocabulary.

(That being said, if you are an OOP developer who's used to using languages like Java, Kotlin, Python, Dart, etc., I was once in your shoes, and I remember that using `return` can help when you first start writing recursive functions. Therefore, perhaps a more gentle way to say this is, "please drop the `return` keyword from your vocabulary as soon as you feel comfortable letting it go.")

Note 2: Using if/then instead

You can also write this function using an `if/then` expression, but because *pattern matching* is such a big part of Scala and FP, I prefer `match` expressions.

Note 3: Can also use `List()`

Because `Nil` is equivalent to `List()` in Scala, you can also write that `case` expression like this:

```
case List() => 0
```

However, most functional programmers use `Nil`, and I'll continue to use `Nil` in this lesson.

Implementing the second sentence in code

The `case` expression I just wrote is a Scala/FP implementation of the first sentence of our algorithm, so let's move on to the second sentence,

which says, "If the list is *not* empty, the result of the algorithm is the combination of (a) the value of its head element, and (b) the sum of its tail elements."

Since the first `case` expression handles the case of an *empty* list, our second `case` expression should handle the case of a *non-empty* list. Furthermore, knowing that we want to pattern-match a `List` and split it into head and tail components, I start writing the second `case` expression like this:

```
case head :: tail => ???
     ------------
```

If you know your `case` expressions, you know that if `sum` is given a list like `List(1,2,3,4)`, the result of this code is that it assigns the value `1` to `head`, and then `tail` is assigned the value `List(2,3,4)`, like this:

```
head = 1
tail = List(2, 3, 4)
```

> (If you don't know your `case` expressions, please refer to the `match` expression lessons in the Scala Cookbook.)

Great, this `case` expression is a start! But, how do we finish it? Once again I go back to the second sentence:

> If the list is *not* empty, the result of the algorithm is the combination of (a) the value of its head element, and (b) the sum of the tail elements.

The "value of its head element" is easy to add to the `case` expression:

```
case head :: tail => head ...
                     ----
```

But then what comes next? The sentence "the value of its head element, and the sum of the tail elements," tells us we'll be adding *something* to `head`:

```
case head :: tail => head + ???
                           -----
```

What are we adding to `head`? *The sum of the list's tail elements.* Hmm, now how can we get the sum of a list of tail elements? Well, what if we happen to have a function nearby that gives us the sum of a list of integer values?:

```
case head :: tail => head + sum(tail)
                            ---------
```

Whoa. That code is a straightforward implementation of the sentence, isn't it?

(I'll pause here to let that sink in.)

If you combine this new `case` expression with the existing code, you get the following complete `sum` function:

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case head :: tail => head + sum(tail)
```

And that is a recursive "sum the integers in a `List`" function in Scala. Notice that there is no need for `var` fields or `for` loops.

> TIP: If it ever feels weird to call the same function you're currently writing, just imagine that you're calling some other function. For example, in this case, instead of calling `sum(tail)`, imagine you're calling some other function named `addAllElements(tail)` or something like that.

Also, if you're new to Scala, it may be easier to read this function if I put the values on the right side of the `=>` symbol on separate lines:

```
def sum(list: List[Int]): Int = list match
    case Nil =>
        0
    case head :: tail =>
        head + sum(tail)
```

If you're not familiar with `match` expressions, the way cases work inside a `match` expression is that the code on the left side of the `=>` symbol is the *pattern* you're matching, and the code on the right side of the `=>` is the code that will be run when `list` matches the current pattern.

A note on those names

If you're new to `case` expressions, it's important to note that the `head` and `tail` variable names in the second `case` expression can be anything you want. I wrote it like this:

```
case head :: tail => head + sum(tail)
```

but I could have written this:

```
case h :: t => h + sum(t)
```

or this:

```
case x :: xs => x + sum(xs)
```

This last example uses variable names that are commonly used with FP, lists, and recursive programming. When working with a list, a single element is often referred to as `x` (pronounced "ex") and multiple elements are referred to as `xs` (pronounced "exes"). It's a way of indicating that `x` is singular and `xs` is plural, like referring to a single "pizza" or mul-

tiple "pizzas." With lists, the head element is definitely singular, while the tail can contain one or more elements. I'll generally use this naming convention in this book.

Proof that `sum` works

To demonstrate that `sum` works, you can clone my (old) RecursiveSum project on Github — which uses ScalaTest to test `sum` — or you can copy the following source code that uses a Scala 3 `main` method application:

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case x :: xs => x + sum(xs)

@main def RecursiveSum =
    val list = List(1, 2, 3, 4)
    val sum = sum(list)
    println(sum)
```

When you run this application you should see the output, `10`. And if you've never written a recursive function before, congratulations!

"That's great," you say, "but *how* exactly did that end up printing `10`?"

To which I say, "Excellent question. Let's dig into that in the next lesson!"

> As I've noted before, I tend to write verbose code that's hopefully easy to understand — especially in books — but you can shrink the last three lines of code to this, if you prefer:
>
> ```
> println(sum(List(1, 2, 3, 4)))
> ```

6

Recursion: How Recursive Function Calls Work

When recursive calls are made to a function like `sum`, you can envision that they "wind up" as `sum` is called repeatedly. You can imagine that the second `case` expression looks like this as `sum` is called over and over again:

```
sum(List(1, 2, 3, 4))                // the 1st call to `sum`
    => 1 + sum(List(2, 3, 4))        // 2nd call
        => 2 + sum(List(3, 4))       // 3rd
            => 3 + sum(List(4))      // 4th
                => 4 + sum(List())   // 5th (final call)
```

An important point to understand about these recursive calls is that just as they wind up as they're called repeatedly, they "unwind" rapidly when the function's end condition is reached.

In the case of the `sum` function, the end condition is triggered when the `Nil` element in a `List` is reached. (Recall that `Nil` is the same as `List()`). When `sum` is passed the `Nil` element as the last element of the `List`, this pattern of the `match` expression is matched:

```
case Nil => 0
     ---
```

Because this line simply returns `0`, there are no more recursive calls to `sum`. As I've mentioned, handling the `Nil` element is the typical way of ending the recursion when you operate on all elements of a `List` in recursive algorithms.

Reminder: Lists end with `Nil`

As I mentioned previously, a very literal way to create a `List` looks like these examples:

```
1 :: 2 :: 3 :: Nil         // List[Int]
'a' :: 'b' :: 'c' :: Nil   // List[Char]
"a" :: "b" :: "c" :: Nil   // List[String]
```

This is a reminder that with ANY Scala `List` you are *guaranteed* that the last `List` element is `Nil`. Therefore, if your algorithm is going to iterate over the entire list, you should use this as your function's end condition:

```
case Nil => ???
```

This is our first clue about how the "unfolding" process works.

> NOTE 1: `Nil` being the last element is a feature of the Scala `List` class. You'll have to change the approach if you work with other sequential collection classes like `Vector`, `ArrayBuffer`, etc.

> NOTE 2: Examples of functions that are built into the collections classes and iterate over every element in a list are `map`, `filter`, `foreach`, `sum`, `product`, and many more. Conversely, examples of functions that *may not* operate on every list element are `take` and `takeWhile`.

Understanding how the `sum` example ran

A good way to understand how the `sum` function example runs is to add `println` statements inside the `case` expressions. First, change the `sum` function to look like this:

```
def sum(list: List[Int]): Int = list match
```

```
    case Nil =>
        println("case1: Nil was matched")
        0
    case head :: tail =>
        println(s"case2: head = $head, tail = $tail")
        head + sum(tail)
```

Now when you call it with a `List(1,2,3,4)` as its input parameter, you'll see this output:

```
case2: head = 1, tail = List(2, 3, 4)
case2: head = 2, tail = List(3, 4)
case2: head = 3, tail = List(4)
case2: head = 4, tail = List()
case1: Nil was matched
```

That output shows that `sum` is called repeatedly until the list is reduced to `List()` (which is the same as `Nil`). When `List()` is passed to `sum`, the first `case` is matched and the recursive calls to `sum` come to an end. (I'll demonstrate this visually in the next lesson.)

The book Land of Lisp[1] states, "recursive functions are 'list eaters,'" and this output shows exactly why that statement is true.

How the recursion works ("going down")

Keeping in mind that `List(1,2,3,4)` is the same as `1::2::3::4::Nil`, you can read the output like this:

1. The first time `sum` is called, the `match` expression sees that the given `List` does NOT match the `Nil` element, so control flows to the second `case` statement.

[1] https://amzn.to/3WRp6Cs

2. The second `case` statement DOES matches the `List` pattern, then splits the incoming list of `1::2::3::4::Nil` into (a) a head element of `1` and the remainder of the list, `2::3::4::Nil`. The remainder — the tail — is then passed into another `sum` function call.

3. A new instance of `sum` receives the list `2::3::4::Nil`. It sees that this list does not match the `Nil` element, so control flows to the second `case` statement.

4. That statement matches the `List` pattern, then splits the list into a head element of `2` and a tail of `3::4::Nil`. The tail is passed as the input parameter to another `sum` call.

5. A new instance of `sum` receives the list `3::4::Nil`. This list does not match the `Nil` element, so control passes to the second `case` statement.

6. The list matches the pattern of the second `case` statement, which splits the list into a head element of `3` and a tail of `4::Nil`. The tail is passed as the input parameter to another `sum` call.

7. A new instance of `sum` receives the list `4::Nil`, sees that it does not match `Nil`, and passes control to the second `case` statement.

8. The list matches the pattern of the second `case` statement, and it's split into a head element of `4` and a tail of `Nil`. The tail is passed to another `sum` function call.

9. The new instance of `sum` receives `Nil` as an input parameter, and sees that it DOES match the `Nil` pattern in the first `case` expression. At this point the first `case` expression is evaluated.

10. The first `case` expression returns the value `0`. This marks the end of the recursive calls.

At this point — when the first `case` expression of this `sum` instance returns `0` — all of the recursive calls "unwind" until the very first `sum` instance returns its answer to the code that called it.

How the unwinding works ("coming back up")

That description gives you an idea of how the recursive `sum` function calls work until they reach the end condition. Here's a description of what happens *after* the end condition is reached:

1. The last `sum` instance — the one that received `List()` — returns `0`. This happens because `List()` matches `Nil` in the first `case` expression.
2. This returns control to the previous `sum` instance. The second `case` expression of that `sum` function has `return 4 + sum(Nil)` as its return value. This is reduced to `return 4 + 0`, so this instance returns `4`. (Note that I *don't* use a `return` statement in the actual code, but I find that the following examples are easier to read when I use `return`.)
3. Again, this returns control to the previous `sum` instance. That `sum` instance has `return 3 + sum(List(4))` as the result of its second `case` expression. You just saw that `sum(List(4))` returns `4`, so this `case` expression evaluates to `return 3 + 4`, or `7`.
4. Control is returned to the previous `sum` instance. Its second `case` expression has `return 2 + sum(List(3,4))` as its result. You just saw that `sum(List(3,4))` returns `7`, so this expression evaluates to `return 2 + 7`, or `9`.
5. Finally, control is returned to the original `sum` function call. Its second `case` expression is `return 1 + sum(List(2,3,4))`. You just saw that `sum(List(2,3,4))` returns `9`, so this call is reduced to `return 1 + 9`, or `10`. This value is returned to whatever code called the first `sum` instance.

Initial visuals of how the recursion works

One way to visualize how the recursive `sum` function calls work — the "going down" part — is shown in Figure 6.1.

```
sum(List(1,2,3,4))
    -> sum(List(2,3,4))
          -> sum(List(3,4))
                -> sum(List(4))
                      -> sum(List())
```

Figure 6.1: How the original `sum` *call leads to another, then to another …*

After that, when the end condition is reached, the "coming back up" part — what I call the unwinding process — is shown in Figure 6.2.

```
                      -> sum(Nil)    // return 0
                -> sum(List(4))      // return 4 + sum(List())       => return 4 + 0 => 4
          -> sum(List(3,4))          // return 3 + sum(List(4))      => return 3 + 4 => 7
    -> sum(List(2,3,4))               // return 2 + sum(List(3, 4))   => return 2 + 7 => 9
sum(List(1,2,3,4))                    // return 1 + sum(List(2, 3, 4)) => return 1 + 9 => 10
```

Figure 6.2: How `sum` *function calls unwind, starting with the last* `sum` *call.*

If this isn't clear, fear not, in the next lesson I'll show a few more visual examples of how this works.

7

Visualizing the Recursive sum Function

Another way to think about recursion is with visual diagrams. To demonstrate this, I'll use the rectangular symbol shown in Figure 7.1 to represent a function.

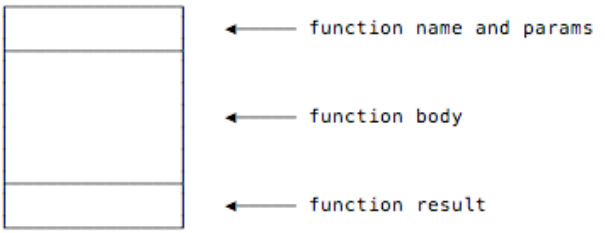


Figure 7.1: This rectangular symbol is used to represent functions in this lesson.

The first step

Using that symbol and a `List(1,2,3)`, Figure 7.2 shows a representation of the first `sum` function call.

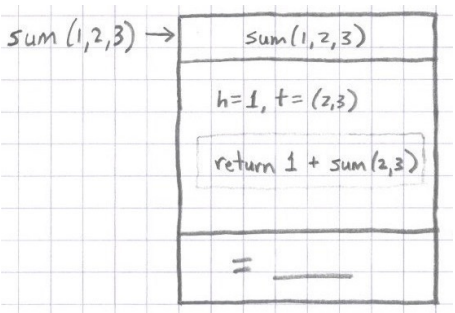


Figure 7.2: A visual representation of the first `sum` *call.*

The top cell in the rectangle indicates that this first instance of `sum` is called with the parameters `1,2,3`. Note that I'm leaving the "`List`" name off of these diagrams to make them more readable.

The body of the function is shown in the middle region of the symbol, and it's shown as `return 1 + sum(2,3)`. As I mentioned before, you don't normally use the `return` keyword with Scala/FP functions, but in this case it makes the diagram more clear. Note that in this example, `h` stands for *head*, and `t` stands for *tail*.

In the bottom region of the symbol I've left room for the final return value of the function. At this time we don't know what the function will return, so for now I just leave that spot empty.

The next steps

For the next step of the diagram, we know that the first `sum` function call receives the parameter list `(1,2,3)`, and its body now calls a new instance of `sum` with the input parameter `sum(2,3)` (or `sum(List(2,3))`, if you prefer). You can imagine the second `case` expression separating the `List` into head (`h`) and tail (`t`) elements, as shown in Figure 7.3.

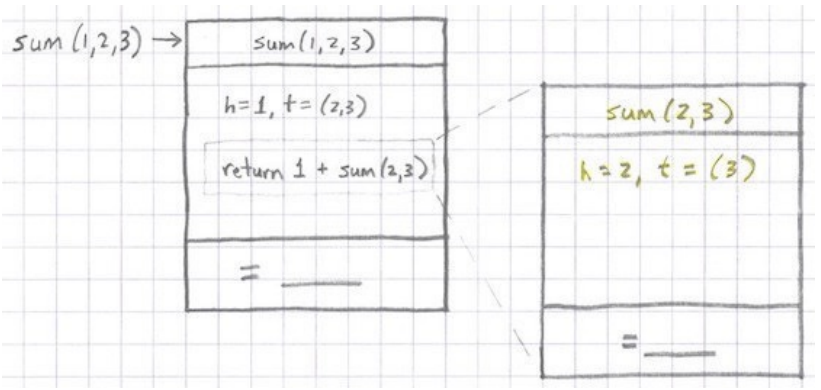


Figure 7.3: The first `sum` *function invokes a second* `sum` *function call.*

Then this `sum` instance makes a recursive call to another `sum` instance, as

shown in Figure 7.4.

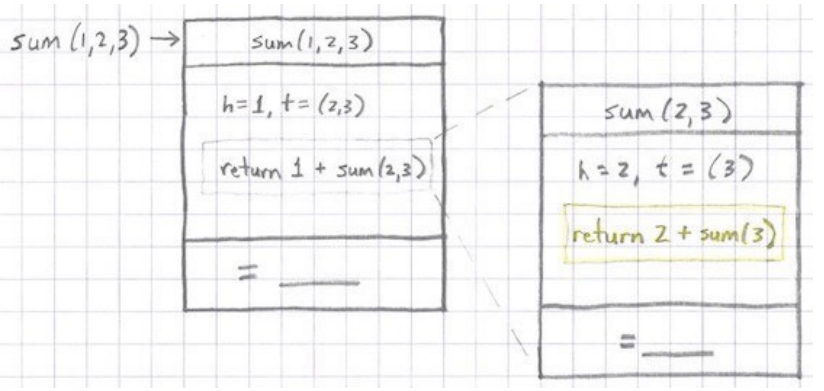


Figure 7.4: The second `sum` *function call begins to invoke the third* `sum` *instance.*

Again I leave the return value of this function empty because I don't know what it will be until its `sum` call returns.

It's important to be clear that these two function calls are completely different instances of `sum`. They have their own input parameter lists, local variables, and return values. It's just as if you had two different functions, one named `sum3elements` and one named `sum2elements`, as shown in Figure 7.5.

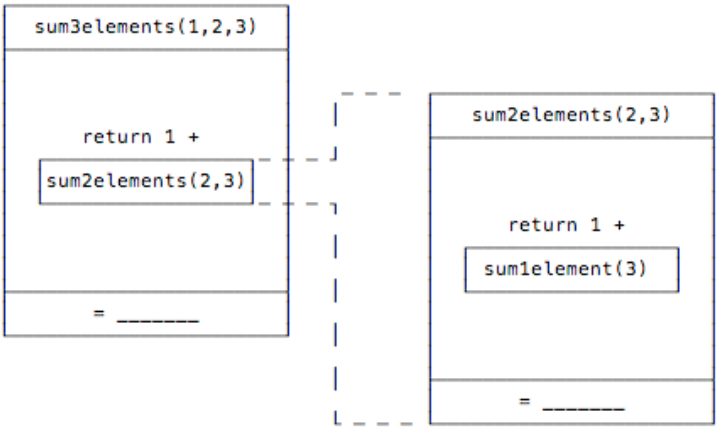


Figure 7.5: One `sum` *function calling another* `sum` *instance is just like calling a different function.*

Just as the variables inside of `sum3elements` and `sum2elements` have completely different scope, the variables in two different instances of `sum` also have completely different scope.

Getting back to the `sum` example, you can now imagine that the next step will proceed just like the previous one, as shown in Figure 7.6.

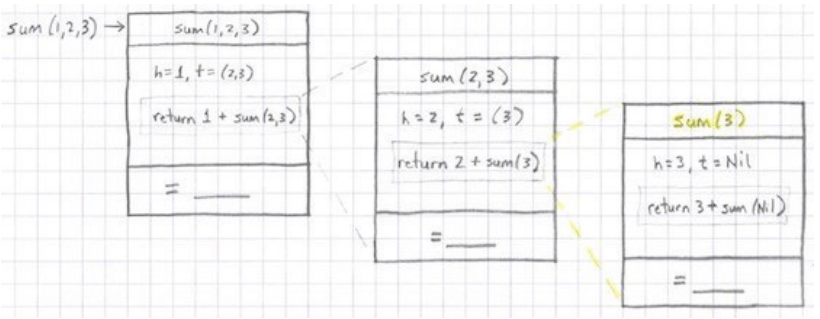


Figure 7.6: The third `sum` *function has now been called as* `sum(List(3)`*, so its head is* `3` *and its tail is* `Nil`.

The last recursive `sum` *call*

Now we're at the point where we make the last recursive call to `sum`. In this case, because `3` was the last integer in the list, a new instance of `sum` is called with the `Nil` value as its input parameter. This is shown in Figure 7.7.

With this last `sum` call, the `Nil` input parameter matches the first `case` expression, and that expression simply returns `0`. So now we can fill in the return value for this function, as shown in Figure 7.8.

Now this `sum` instance returns `0` back to the previous `sum` instance, as shown in Figure 7.9.

The result of this function call is `3 + 0` (which is `3`), so you can fill in its return value, and then flow it back to the previous `sum` call. This is

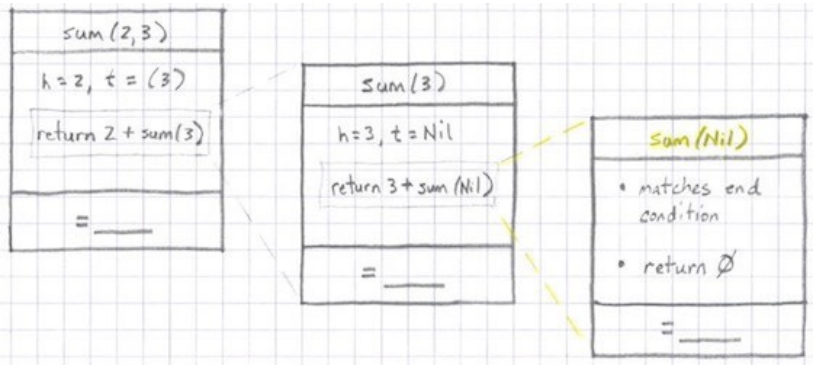


Figure 7.7: `Nil` *is passed into the final* `sum` *function call.*

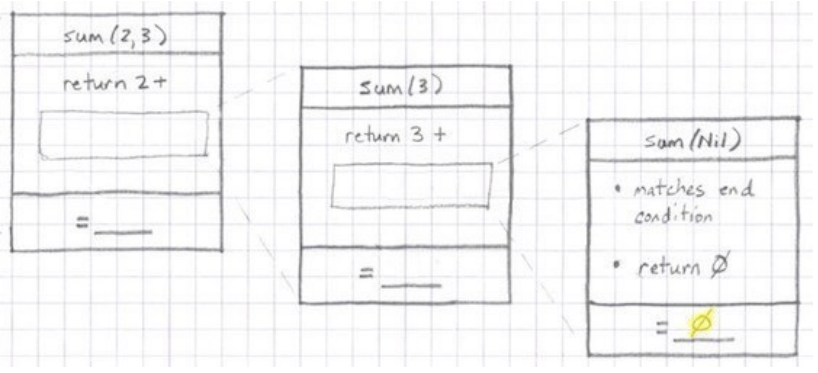


Figure 7.8: The return value of the last `sum` *call is* `0`.

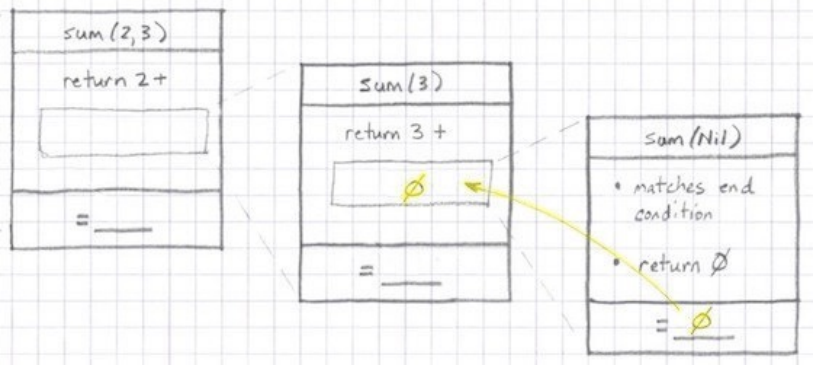


Figure 7.9: `0` *is returned back to the previous* `sum` *call.*

shown in Figure 7.10.

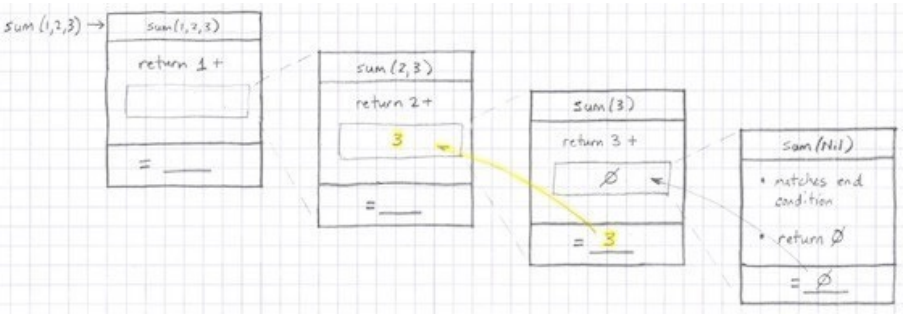


Figure 7.10: The third `sum` *call returns to the second.*

The result of this function call is `2 + 3` (`5`), so that result can flow back to the previous function call, as shown in Figure 7.11.

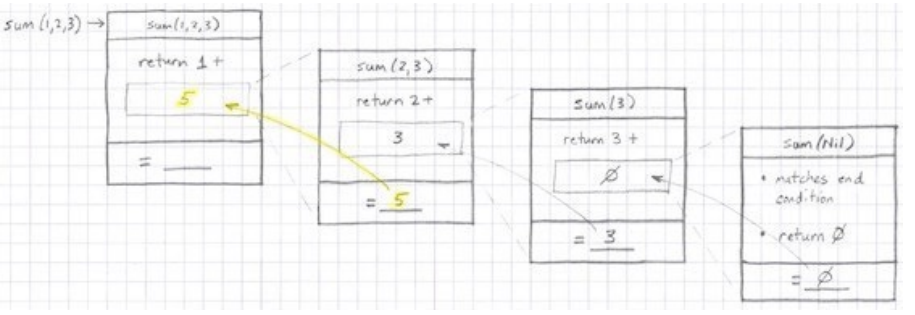


Figure 7.11: The second `sum` *call returns to the first.*

Finally, the result of this `sum` instance is `1 + 5` (`6`). This was the first `sum` function call, so it returns the value `6` back to whoever called it, as shown in Figure 7.12.

> NOTE: There are four recursive calls because there are four elements in the list: the values `1`, `2`, and `3`, as well as the trailing `Nil` element.

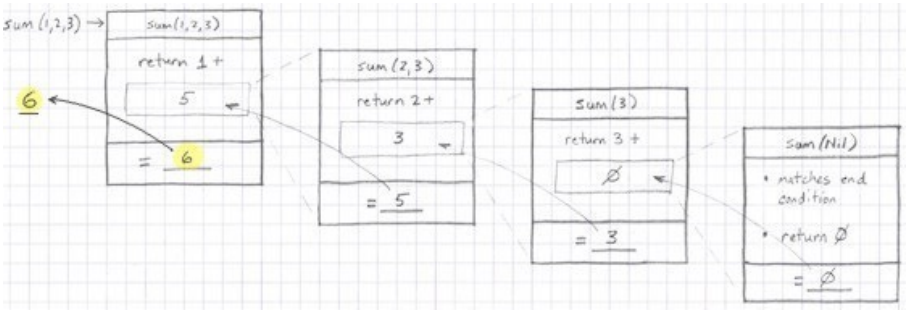


Figure 7.12: The first `sum` *call returns to the final result.*

Other visualizations

There are other ways to draw recursive function calls. Another nice approach is to use a modified version of a UML "Sequence Diagram," as shown in Figure 7.13. Note that in this diagram, "time" flows from the top to the bottom.

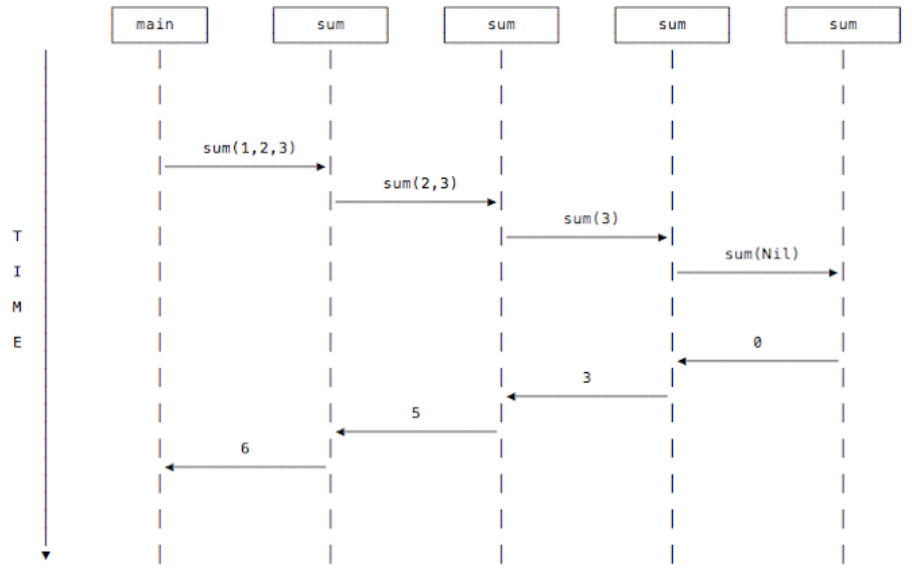


Figure 7.13: The `sum` *function calls can be shown using a UML Sequence Diagram.*

This diagram shows that the `main` method calls `sum` with the parameter `List(1,2,3)`, where I again leave off the `List` part; it calls `sum(2,3)`, and so on, until the `Nil` case is reached, at which point the return values flow

back from right to left, eventually returning `6` back to the `main` method.

You can write the return values like that, or with some form of the function's equation, as shown in Figure 7.14.

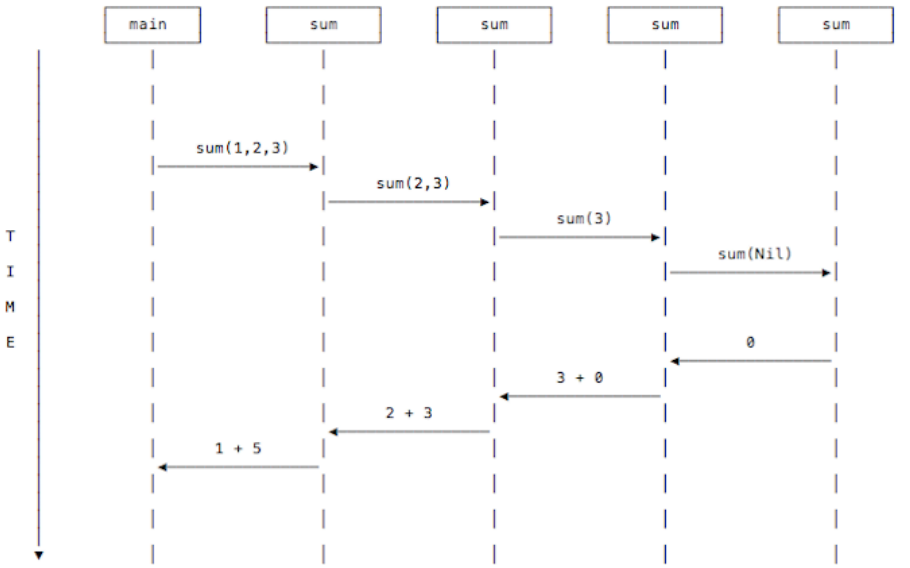


Figure 7.14: Writing the function return values as "head + sum(tail)" equations.

Personally, I use whatever diagram seems to help the most.

Summary

Those are some visual examples of how recursive function calls work. If you find yourself struggling to understand how recursion works, I hope these diagrams are helpful.

8

Recursion: A Conversation Between Two Developers

As an homage to one of my favorite Lisp books — an early version of what is now The Little Schemer[1] — this lesson shows a little "question and answer" interaction that you can imagine happening between two Scala programmers.

Given this `sum` function:

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case head :: tail => head + sum(tail)
```

I hope this "conversation" will help drive home some of the points about how recursion works:

Person 1	Person 2
What is this? `val xs = List(1,2,3,4)`	An expression that defines a `List[Int]`, which in this case contains the integers `1` through `4`. The expression binds that list to the immutable variable `xs`.
And what is this? `xs.head`	The first element of the list `xs`, which is `1`.

[1] https://amzn.to/3WV8N7S

Person 1	Person 2
How about this? `xs.tail`	That's the remaining elements in the list `xs`, which is `List(2,3,4)`.
How about this: `xs.tail.head`	It is the number `2`.
How did you come up with that?	`xs.tail` is `List(2,3,4)`, and `List(2,3,4).head` is the first element of that list, or `2`.
How about this: `xs.tail.tail`	That's `List(3,4)`.
Explain, please.	`xs.tail` is `List(2,3,4)`, and then `List(2,3,4).tail` is `List(3,4)`.
Are you ready for more?	Yes, please.
Given the definition of our `sum` function, explain the first step in: `sum(List(1,2,3))`.	The `sum` function receives `List(1,2,3)`. This does not match the `Nil` case, but does match the second case, where `head` is assigned to `1` and `tail` is `List(2,3)`.
Then what happens?	A new instance of `sum` is called with the parameter `List(2,3)`.
And then?	The new instance of `sum` receives the input parameter `List(2,3)`. This does not match the `Nil` case, but does match the second case, where `head` is assigned to `2` and `tail` is `List(3)`.

Person 1	Person 2
Please continue.	A new instance of `sum` is called with the parameter `List(3)`.
Go on.	The new `sum` instance receives `List(3)`. This does not match the `Nil` case, but does match the second case, where `head` is assigned to `3` and `tail` is `List()`.
Don't stop now.	`sum` is called with the parameter `List()`.
What happens inside this instance of `sum`?	It receives `List()`. This is the same as `Nil`, so it matches the first case.
Cool. Something different. Now what happens?	That `case` returns `0`.
Ah, finally a return value!	You're telling me.
Okay, so now what happens?	This ends the recursion, and then the recursive calls unwind, as described in the previous lesson.

9

Recursion: Thinking Recursively

"To understand recursion, one must first understand recursion."

Stephen Hawking

Goal: The recursive pattern

This lesson has one primary goal: to show that the thought process followed in writing the `sum` function follows a common recursive programming "pattern." Indeed, when you write recursive functions you'll generally follow the three-step process shown in this lesson.

I don't want to make this too formulaic, but the reality is that if you follow these steps in your thinking, it will make it easier to write recursive functions, especially when you first start.

The general recursive thought process (the "three steps")

As I mentioned in the previous lessons, when I sit down to write a recursive function, I think of three things:

- What is the function signature?
- What is the end condition for this algorithm?
- What is the actual algorithm? For example, if I'm processing all of the elements in a `List`, what does my algorithm do when the

> function receives a non-empty `List` — i.e., what do I want to do with the head and tail elements?

Let's take a deep dive into each step in the process to make more sense of these descriptions.

Step 1: What is the function signature?

Once I know that I'm going to write a recursive function — or any pure function, for that matter — the first thing I ask myself is, "What is the *type signature* of this function?"

I find that if I can describe a function verbally, I can quickly figure out (a) the parameters that will be passed into it and (b) what the function will return. In fact, if you *don't* know these things, you're not ready to write the function yet, are you?

> "If I were given one hour to save the planet, I'd spend 59 minutes defining the problem, and one minute resolving it."
>
> ~ Albert Einstein

The sum function

In the `sum` function, the algorithm is to "add all of the integers in a given list together to return a single integer result." Therefore, because I know the function takes a list of integers as its input, I can start sketching the function signature like this:

```
def sum(list: List[Int]) ...
              ---------
```

Because the description also tells me that the function returns a single `Int` result, I add the function's return type:

```
def sum(list: List[Int]): Int = ???
                            ---
```

This is the Scala way to say that "the `sum` function takes a list of integers and returns an integer result," which is what I want. In FP, sketching the function signature is often half of the battle, so this is actually a big step.

Step 2: How will this algorithm end?

When writing a recursive function, the next thing I usually think is, "How will this algorithm end? What is its *end condition*?"

Because a recursive function like `sum` keeps calling itself over and over, it's of the utmost importance that there is an end case. If a recursive algorithm doesn't have an end condition, it will keep calling itself as fast as possible until either (a) your program crashes with a `StackOverflowError`, or (b) your computer's CPU gets extraordinarily hot. Therefore, I reiterate this tip:

> TIP: Always have an end condition, and write it as soon as possible.

In the `sum` algorithm you know that you have a `List`, and you want to march through the entire `List` to add up the values of all of its elements. You may not know it at this point in your recursive programming career, but right away this statement is a big hint about the end condition. Because:

- you know that you're working with a `List`,
- you want to operate on the *entire* `List`, and
- a `List` ends with the `Nil` element

you can start to write the end condition `case` expression like this:

```
case Nil => ???
```

Because the `Nil` element is to a `List` as a caboose is to a train, you're *guaranteed* that it is always the last element of any `List`.

> Conversely, if your algorithm will NOT iterate over the *entire* `List`, the end condition will be different than this.

Now the next question is, "What should this end condition return?"

A clue here is that the function signature states that it returns an `Int`. Therefore, you know that this end condition must return an `Int` of some sort. But what `Int`?

Because this is a "sum" algorithm, you also know that you don't want to return anything that will affect the sum. Hmm … what `Int` can you return when the `Nil` element is reached that won't affect the sum of a list of integers?

The answer is `0`.

> NOTE: There's a general rule about this thought process, and I'll share it shortly.

Given that answer, I can update the first `case` condition:

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case ???
```

That condition states that if the function receives an empty `List` — denoted by `Nil` — the function will return `0`.

Now we're ready for the third step.

ASIDE: Identity (Neutral) Elements

I'll expand more on the point of returning `0` in this algorithm in the coming lessons, but for now it may help to know that there's a mathematical theory involved in this decision. Per this Wikipedia page[a], "In mathematics, an identity element (or neutral element) of a binary operation operating on a set is an element of the set that leaves unchanged every element of the set when the operation is applied."

Put in simpler words, in our example (a) our *set* is a `List[Int]`, and (b) our *operation* is our sum algorithm. Given that combination of *set+operation*, the identity element is the value `0`, because when you add `0` to any other integer value, it has no effect on that value. (As alluded to in the description, the value `0` is "neutral" for this combination of set+operation.)

To help drive this home, here are a few other identity elements for different set+operation combinations:

1) Imagine that you want to write a "product" algorithm for a list of integers. What would you return for the end condition in this case?

As a reminder, we need some *neutral value* so that when any integer value is multiplied by it, the resulting value is the same as the initial value. That is, given this equation:

```
a = identityElement * 100
```

the requirement is that `a` must also be `100`. Therefore, what must the value of `identityElement` be?

The correct answer is `1`. This is because when any integer value is multiplied by `1`, the result is the same as the original integer. (The number `1` is the neutral element for the set+operation combination of (a) a `List[Int]` combined with (b) a product algorithm.)

> 2) Next, imagine that you're writing a concatenation algorithm for a `List[String]`. What would your return value be for the end condition in this case? (i.e., what is a neutral element for a string?)
>
> As a concrete example, what must `identityElement` be in the following equation so that `a` is also `"foo"`?
>
> ```
> a = identityElement + "foo"
> ```
>
> The correct answer is a blank space. For the combination of (a) a list (or set) of strings, and (b) an addition algorithm, a blank space has no effect on the final result.
>
> [a] https://en.wikipedia.org/wiki/Identity_element

Step 3: What is the algorithm?

Getting back to our `sum` algorithm, now that you've defined the function signature and the end condition, the final question is, "What is the algorithm at hand?"

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case ???        // TODO: what is the algorithm here?
```

In our situation — where our algorithm operates on all of the elements in a `List` and the first `case` condition handles the "empty list" case — this question becomes, "What should my function do when it receives a *non-empty* `List`?"

The answer for a "sum" function is that it should *add* all of the elements in the list.

> Similarly, the answer for a "product" algorithm is that it should multiply all of the list elements.

The sum algorithm

To create the solution, I go back to the original statement of the `sum` algorithm:

> "The sum of a list of integers is the sum of the *head* element, plus the sum of the *tail* elements."

A common way to write the *pattern* for this `case` expression is this:

```
case head :: tail => ???
     ------------
```

This pattern is the Scala way to say, "`head` will be bound to the value of the first element in the `List`, and `tail` will contain all of the remaining elements in the `List`."

Because my description of the algorithm states that the sum is "the sum of the *head* element, plus the sum of the *tail* elements," I now start to write the algorithm that goes on the right side of the `=>` symbol. I start by adding the head element:

```
case head :: tail => head + ...
                     ----
```

and then I add this code to represent "plus the sum of the tail elements":

```
case head :: tail => head + sum(tail)
                            -----------
```

Now that we have the function signature, the end condition, and the list-processing algorithm, we have the complete function:

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case head :: tail => head + sum(tail)
```

Before I move on, if you're new to Scala it can help to see the return value on the right side of the => symbol on its own line:

```
def sum(list: List[Int]): Int = list match
    case Nil =>
         0
    case head :: tail =>
         head + sum(tail)
```

That helps by separating the pattern-matching on the left side of the => from the resulting value on its right side.

Also, almost nobody in the Scala community uses the `return` keyword, but if you're coming to Scala from an OOP language, I understand that adding it in can make your code easier to read initially:

```
def sum(list: List[Int]): Int = list match
    case Nil =>
         return 0
    case head :: tail =>
         return head + sum(tail)
```

But in the long run, remember that pure, algebraic functions don't "return" a value; they *evaluate to a result* (so drop the `return` keyword as soon as you can).

Naming conventions

As I noted in the previous lessons, when FPers work with lists, they often prefer to use the variable name `x` to refer to a single element and `xs` to refer to multiple elements, so you'll also see recursive functions that process lists use these variable names:

```
def sum(list: List[Int]): Int = list match
```

```
case Nil => 0
case x :: xs => x + sum(xs)
```

But you don't have to use any of those names; use whatever names work best for you.

The last two steps are iterative

In practice, the first step — sketching the function signature — is almost always the first step in the process. As I mentioned, you can't really write a function if you don't know what the inputs and output will be.

But the last two steps — defining the end condition and writing the algorithm — are interchangeable, and even iterative. For instance, if you're working on a `List` and you want to do something for *every* element in the list, you know the end condition will occur when you reach the `Nil` element. But if you're NOT going to operate on the *entire* list, or if you're working with something other than a `List`, it can help to bounce back and forth between the end case and the main algorithm until you come to the solution.

Key points

As a recap of the key concepts in this lesson, when I sit down to write a recursive function, I generally think of three things:

- What is the function signature?
- What is the end condition for this algorithm?
- What is the main algorithm?

To solve the problem I almost always write the function signature first, and after that I usually write the end condition next, though the last two steps can also be an iterative process.

Another key point is knowing the identity (neutral) element for the combination of set+algorithm that you're writing.

As a final note for this lesson, I thought about showing how to write a `map` function using recursion, but as I thought about it, I realized you don't need to use recursion for that algorithm, you just need a `for` expression. I show how to do that in my blog post, How to Write a 'map' Function in Scala[1]

[1] https://alvinalexander.com/scala/fp-book/how-to-write-scala-map-function

10

JVM Stacks, Stack Frames, and Stack Overflow Errors

For functions without deep levels of recursion, there's nothing wrong with the algorithms shown in the previous lessons. I use this simple, basic form of recursion when I know that I'm working with limited data sets. But in applications where you don't know how much data you might be processing, it's important that your recursive algorithms are *tail-recursive*, otherwise you'll get a nasty `StackOverflowError`.

For instance, if you run the `sum` function from the previous lessons with a larger list, like this:

```
@main def RecursiveSum =
    def sum(list: List[Int]): Int = list match
        case Nil => 0
        case x :: xs => x + sum(xs)

    val list = List.range(1, 100_000)   // MUCH MORE DATA
    val x = sum(list)
    println(x)
```

you'll get a `StackOverflowError`, which is *really* counter to our desire to write great, bulletproof, functional programs.

> The actual number of integers in a list needed to produce a `StackOverflowError` with this function will depend on the `java` command-line settings you use, but the last time I checked the default Java stack size it was 1,024 kb — yes, 1,024 *kilobytes* — just over one million *bytes*. That's not much

> RAM to work with. I write more about this at the end of this lesson, including how to change the default stack size with the `java` command's `-Xss` parameter.

I'll cover tail recursion in the next lesson, but in this lesson I want to discuss the *JVM stack* and *stack frames*. If you're not already familiar with these concepts, this discussion will help you understand why this code results in an exception. It can also help you debug "stack traces" in general.

What is a "Stack"?

To understand the potential "stack overflow" problem of recursive algorithms, you need to understand what happens when you write recursive algorithms.

The first thing to know is that in all computer programming languages there is this thing called *the stack*, also known as the "call stack."

Official Java/JVM "stack" definition

Oracle provides the following description of the stack and stack frames as they relate to the JVM:

> "Each JVM thread has a private Java virtual machine stack, created at the same time as the thread. A JVM stack stores frames, also called 'stack frames.' A JVM stack is analogous to the stack of a conventional language such as C — it holds local variables and partial results, and plays a part in method invocation and return."

Given that description, you can visualize that a single stack has a pile of stack frames that look like Figure 10.1.

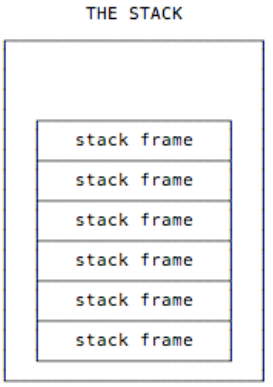


Figure 10.1: A single stack has a pile of stack frames.

As that description mentions, each thread has its own stack, so in a multi-threaded application there are multiple stacks, and each stack has its own stack of frames, as shown in Figure 10.2.

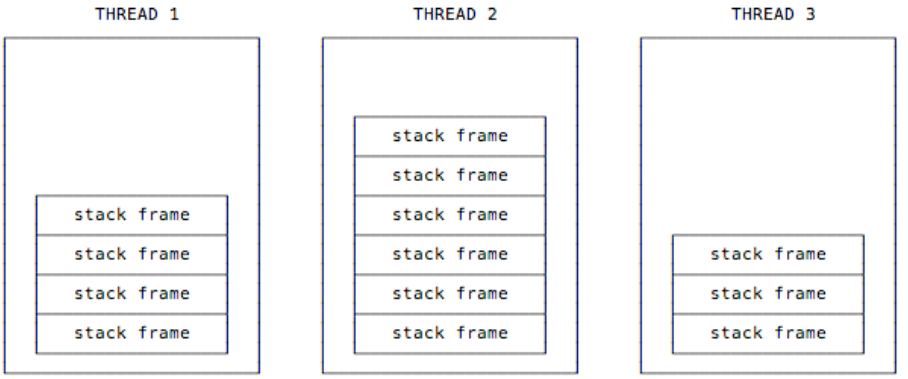


Figure 10.2: Each thread has its own stack.

The Java stack

To explain the *stack* a little more, all of the following quoted text comes from the free, online version of a book titled, Inside the Java Virtual Machine, by Bill Venners (who is also known for creating the ScalaTest

testing framework). (I edited the text slightly to include only the portions relevant to stacks and stack frames.)

> "When a new thread is launched, the JVM creates a new stack for the thread. A Java stack stores a thread's state in discrete frames. *The JVM only performs two operations directly on Java stacks: it pushes and pops frames.*
>
> The method that is currently being executed by a thread is the thread's current method. The stack frame for the current method is the current frame. The class in which the current method is defined is called the current class, and the current class's constant pool is the current constant pool. As it executes a method, the JVM keeps track of the current class and current constant pool. When the JVM encounters instructions that operate on data stored in the stack frame, it performs those operations on the current frame.
>
> *When a thread invokes a Java method, the JVM creates and pushes a new frame onto the thread's stack.* This new frame then becomes the current frame. As the method executes, it uses the frame to store parameters, local variables, intermediate computations, and other data."

TIP: As the previous paragraph implies, each instance of a method has its own stack frame. Therefore, when you see the term "stack frame," you can think, "all of the stuff a method instance needs."

What is a "Stack Frame"?

The same chapter in that book describes the "stack frame" as follows: "The stack frame has three parts: local variables, operand stack, and frame data."

You can visualize that as shown in Figure 10.3.

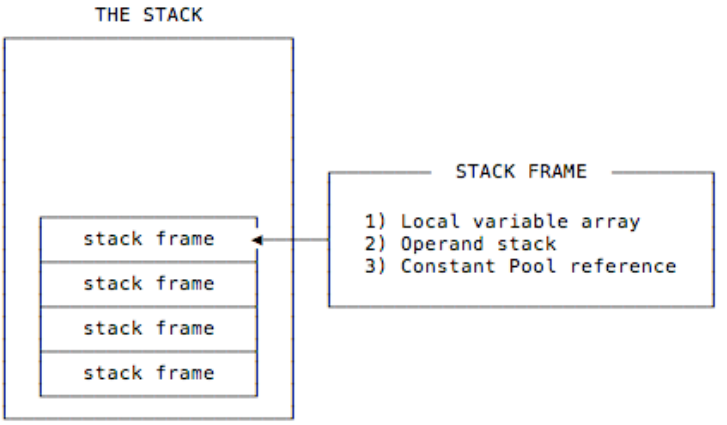


Figure 10.3: Each stack frame has three parts

The book continues:

> "The sizes of the local variables and operand stack, which are measured in words, depend upon the needs of each individual method. These sizes are determined at compile time and included in the class file data for each method."

That's important: *the size of a stack frame varies depending on the local variables and operand stack.* The book describes that size like this:

> "When the JVM invokes a method, it checks the class data to determine the number of *words* required by the method in the local variables and operand stack. It creates a stack frame of the proper size for the method and pushes it onto the stack."

Word size, operand stack, and constant pool

These descriptions introduce the phrases word size, operand stack, and constant pool. Here are definitions of those terms:

First, *word size* is a unit of measure. From Chapter 5 of the same book, the word size can vary in JVM implementations, but it must be at least 32 bits so it can hold a value of type long or double.

Next, the *operand stack* is defined here on oracle.com, but as a word of warning, that definition gets into machine code very quickly. For instance, it shows how two integers are added together with the `iadd` instruction. You are welcome to dig into those details, but for our purposes, a simple way to think about the operand stack is that it's memory (RAM) that is used as a working area inside a stack frame.

The Java *run-time constant pool* is defined at this oracle.com page, which states, "A run-time constant pool … contains several kinds of constants, ranging from numeric literals known at compile-time, to method and field references that must be resolved at run-time. The run-time constant pool serves a function similar to that of a symbol table for a conventional programming language, although it contains a wider range of data than a typical symbol table."

Summary to this point

You can summarize what you've learned about stacks and stack frames like this:

- Each JVM thread has a private stack, created at the same time as the thread.
- A stack stores frames, also called *stack frames*.
- A stack frame is created every time a new method is called.

We can also say this about what happens when a Java/Scala/JVM method is invoked:

- When a method is invoked, a new stack frame is created to contain information about that method.

- Stack frames can have different sizes, depending on the method's parameters, local variables, and algorithm.
- As the method is executed, the code can only access the values in the current stack frame, which you can visualize as being the top-most stack frame.

As it relates to recursion, that last point is important. As a function like our `sum` function works on a list (such as `List(1,2,3)`), information about that instance of `sum` is in the top-most stack frame, and that instance of `sum` can't see the data of other instances of the `sum` function. This is how what appears to be a single, local variable — like the values `head` and `tail` inside of `sum` — can seemingly have many different values at the same time: they're all maintained in different stack frames.

One last resource on the stack and recursion

To be thorough, I want to share one last description of the stack (and the heap) that has specific comments about recursion. The discussion in Figure 10.4 comes from a book named Algorithms, by Sedgewick and Wayne.

There are two important lines in this description that relate to recursive algorithms:

- When the method returns, that information is popped off the stack, so the program can resume execution just after the point where it called the method.
- recursive algorithms can sometimes create extremely deep call stacks and exhaust the stack space.

What we now know

From all of these discussions I hope you can see the potential problem of recursive algorithms:

SERIOUS STACK SPACE

Normally a computer allocates two areas of memory for a program: the stack and the heap.

The stack is used to store information about method calls. When a piece of code calls a method, information about the call is placed on the stack. When the method returns, that information is popped off the stack, so the program can resume execution just after the point where it called the method. (The stack is the same kind of stack described in Chapter 5.) The list of methods that were called to get to a particular point of execution is called the *call stack*.

The heap is another piece of memory that the program can use to create variables and perform calculations.

Typically the stack is much smaller than the heap. The stack usually is large enough for normal programs because your code typically doesn't include methods calling other methods to a very great depth. However, recursive algorithms can sometimes create extremely deep call stacks and exhaust the stack space, causing the program to crash.

For this reason, it's important to evaluate the maximum depth of recursion that a recursive algorithm requires in addition to studying its run time and memory requirements.

Figure 10.4: A discussion of the JVM stack and heap

- When a recursive function calls itself, information for the new instance of the function is pushed onto the stack in the form of a new stack frame.
- Each time the function calls itself, another copy of the function information is pushed onto the stack. With each recursive call, a new stack frame is added to the stack.
- As a result, more and more memory that is allocated to the stack is consumed as the function recurses. If the `sum` function calls itself a million times, a million stack frames are created.
- Because the JVM stack size is relatively small, it's easy for a recursive function to consume all of this memory, which results in a `StackOverflowError`.

11

A Visual Look at Stacks and Frames

Given the background information of the previous lesson, let's take a visual look at how the JVM stack and stack frames work by going back to our recursive `sum` function from the previous lessons.

Before the `sum` function is initially called, the only thing on the call stack is the application's `main` method, as shown in Figure 11.1.

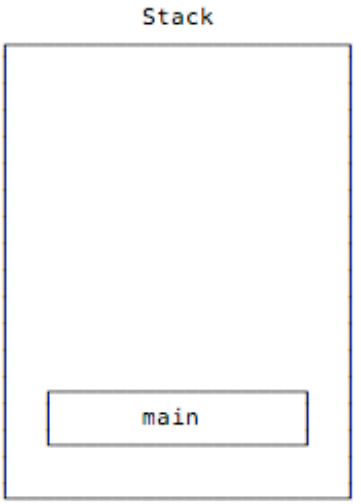


Figure 11.1: `main` *is the only method on the call stack before* `sum` *is called.*

Then `main` calls `sum` with `List(1,2,3)`, which I show in Figure 11.2 without the "List" to keep things simple.

The data that's given to `sum` matches its second `case` expression, and in my pseudocode, that expression evaluates to this:

```
return 1 + sum(2,3)
```

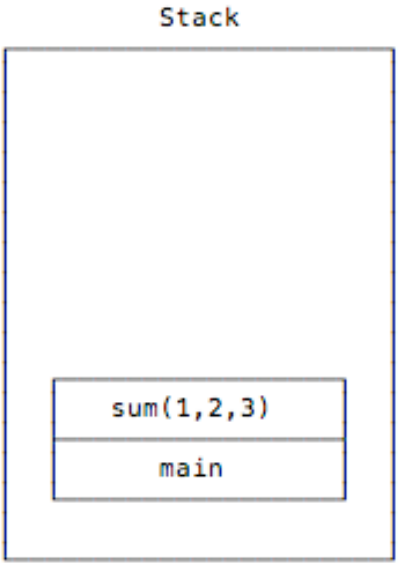


Figure 11.2: The first `sum` *call is added to the stack.*

So now a new instance of `sum` is called with `List(2,3)`, and the stack looks as shown in Figure 11.3.

Inside this `sum` call, the second `case` expression is matched, and the right side of the `=>` symbol evaluates to this:

```
return 2 + sum(3)
```

At this point a new instance of `sum` is called with the input parameter `List(3)`, and the stack looks like Figure 11.4.

Once again the second `case` expression is matched, and the right side of the `=>` symbol evaluates to this:

```
return 3 + sum(Nil)
```

Finally, another instance of `sum` is called with the input parameter `Nil` — also known as `List()` — and the stack now looks like Figure 11.5.

This time, when `sum(Nil)` is called, the first `case` expression is matched:

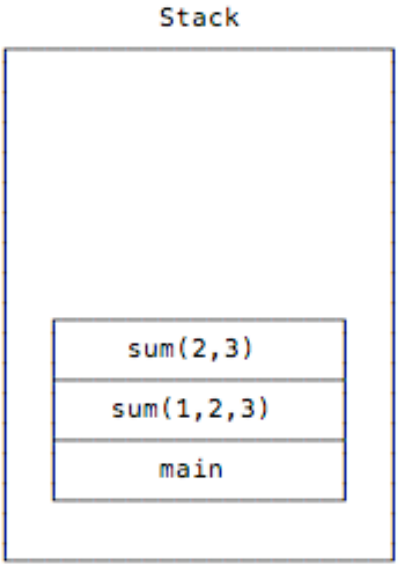


Figure 11.3: The second `sum` *call is added to the stack.*

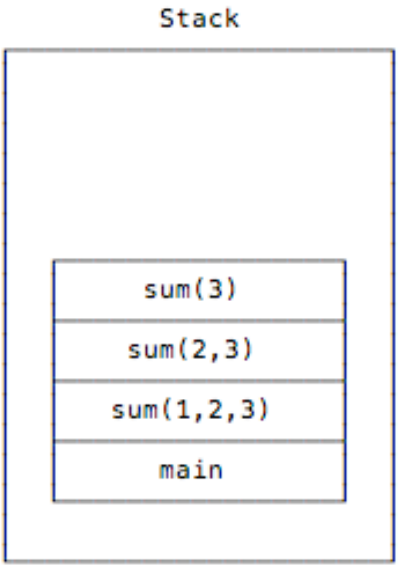


Figure 11.4: The third `sum` *call is added to the stack.*

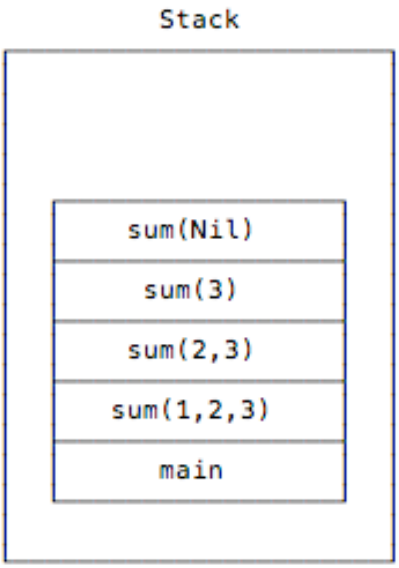


Figure 11.5: The fourth (and final) `sum` *call is added to the stack.*

```
case Nil => 0
```

That pattern match causes this `sum` instance to return `0`, and when it does, the call stack unwinds and the stack frames are popped off of the stack, as shown in the series of images in Figure 11.6.

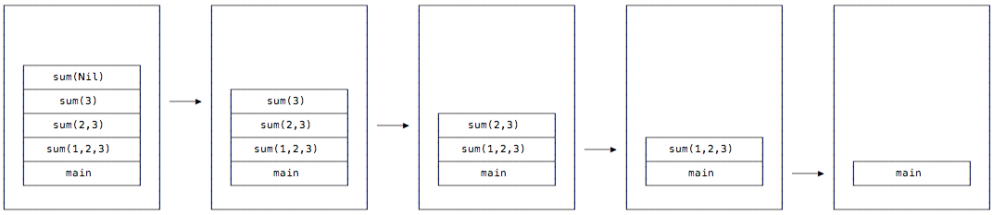


Figure 11.6: In the unwinding of the call stack, the stack frames are popped off the stack as each function yields its result.

In this process, as each `sum` call returns its result, its frame is popped off of the stack, and when the recursion completely ends, the `main` method is the only frame left on the call stack. (The value `6` is also returned by the first `sum` invocation to the place where it was called in the `main` method.)

I hope that gives you a good idea of how recursive function calls are pushed-on and popped-off the JVM call stack.

Manually dumping the stack with the sum example

If you want to explore this in code, you can also see the series of `sum` stack calls by modifying the `sum` function. To do this, add the lines of code shown to the `Nil` case to print out stack trace information when that `case` is reached:

```
def sum(list: List[Int]): Int = list match
    case Nil =>
        // manually create a stack trace and then print it
        val stackTraceAsArray =
            Thread.currentThread.getStackTrace
        stackTraceAsArray.foreach(println)
        // return 0 as before
        0
    case x :: xs => x + sum(xs)
```

Now, if you call `sum` with a list that goes from 1 to 5:

```
val list = List.range(1, 5)
sum(list)
```

you'll get this output when the `Nil` case is reached:

```
java.base/java.lang.Thread.getStackTrace(Thread.java:1602)
rs$line$8$.sum(rs$line$8:5)
rs$line$8$.sum(rs$line$8:10)
rs$line$8$.sum(rs$line$8:10)
rs$line$8$.sum(rs$line$8:10)
rs$line$8$.sum(rs$line$8:10)
```

While that output isn't too exciting, it shows that when the stack dump

is manually triggered when the `Nil` case is reached, the `sum` function is on the stack five times. You can verify that this is correct by repeating the test with a `List` that has three elements, in which case you'll see the `sum` function referenced only three times in the output:

```
java.base/java.lang.Thread.getStackTrace(Thread.java:1602)
rs$line$8$.sum(rs$line$8:5)
rs$line$8$.sum(rs$line$8:10)
rs$line$8$.sum(rs$line$8:10)
```

Clearly the `sum` function is being added to the stack over and over again, once for each call.

> I encourage you to try this on your own to become comfortable with what's happening.

Summary: Our current problem with "basic recursion"

I hope this little dive into the JVM stack and stack frames helps to explain our current problem with "basic recursion." As mentioned, if I try to pass a `List` with 10,000 elements into the current recursive `sum` function, it will generate a `StackOverflowError`. Because we're trying to write bulletproof programs, this isn't good.

What's next

Now that we looked at (a) basic recursion with the `sum` function, (b) how that works with stacks and stack frames in the last two lessons, and (c) how basic recursion can throw a `StackOverflowError` with large data sets, the next lesson shows how to fix these problems with something called "tail recursion."

See also

- Chapter 5 of Inside the Java Virtual Machine, by Bill Venners is an excellent resource. You may not need to read anything more than the content at this URL.
- Chapter 2 of Oracle's JVM Specification is also an excellent resource.
- Here's an article I wrote about the differences between the stack and the heap a long time ago.

One More Thing: Viewing and Setting the JVM Stack Size

"Well," you say, "these days computers have crazy amounts of memory. Why is this such a problem?"

According to this Oracle document, with Java 6 the default stack size was very low: 1,024k on both Linux and Windows.

I encourage you to check the JVM stack size on your favorite computing platform(s). One way to check it is with a command like this on a Unix-based system:

```
java -XX:+PrintFlagsFinal -version | grep -i stack
```

When I do this on my current Mac OS X system, I see that the `ThreadStackSize` is `1024`. I dug through this oracle.com documentation to find that this "1024" means "1,024 Kbytes".

It's important to know that you can also control the JVM stack size with the `-Xss` command line option:

```
$ java -Xss 1M ... (the rest of your command line here)
```

That command sets the stack size to one megabyte. You specify the mem-

ory size attribute as `m` or `M` after the numeric value to get megabytes, as in `1m` or `1M` for one megabyte.

> Use `g` or `G` to specify the size in gigabytes, but if you're trying to use many MB or GB for the stack size, you're doing something wrong. You may need this gigabytes option for the `Xmx` option, but you should never need it for this `Xss` attribute.

The `Xss` option can help if you run into a `StackOverflowError`, BUT, the next lesson on *tail recursion* is intended to help you from ever needing this command line option.

More JVM memory settings

As a final note, you can find more options for controlling Java application memory use by looking at the output of the `java -X` command:

```
$ java -X
```

If you dig through the output of that command, you'll find that the command-line arguments specifically related to Java application memory use are:

```
-Xms  set initial Java heap size
-Xmx  set maximum Java heap size
-Xss  set java thread stack size
```

You can use these parameters on the java command line like this:

```
java -Xms64m -Xmx1G myapp.jar
```

As before, valid memory values end with `m` or `M` for megabytes, and `g` or `G` for gigabytes:

```
-Xms64m or -Xms64M
-Xmx1g or -Xmx1G
```

12

Tail-Recursive Algorithms

"Tail recursion is its own reward."

From the "Functional" cartoon on xkcd.com.

Goal

The goal of this lesson is to solve the problem shown in the previous lessons: Simple recursion creates a series of stack frames, and for algorithms that require deep levels of recursion, this creates a `StackOverflowError` (and crashes your program).

Tail recursion to the rescue

Although the previous lesson showed that algorithms with deep levels of recursion can crash with a `StackOverflowError`, all is not lost. With Scala you can work around this problem by making sure that your recursive functions are written in a tail-recursive style.

A *tail-recursive function* is just a function whose *very last action* is a call to itself. When you write your recursive function in this way, the Scala compiler can optimize the resulting JVM bytecode *so that the function requires only one stack frame* — as opposed to one stack frame for each level of recursion!

On this Stack Overflow page, Martin Odersky (creator of the Scala language) explains tail-recursion in Scala:

> "Functions which call themselves as their last action are called tail-recursive. The Scala compiler detects tail recursion and replaces it with a jump back to the beginning of the function, after updating the function parameters with the new values … as long as the last thing you do is calling yourself, it's automatically tail-recursive (i.e., optimized)."

But that sum function looks tail-recursive to me …

"Hmm," you might say, "if I understand Mr. Odersky's quote, the `sum` function you wrote at the end of the last lesson sure looks tail-recursive to me":

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case x :: xs => x + sum(xs)
                        -------
```

"Isn't the 'last action' a call to itself, making it tail-recursive?"

If that's what you're thinking, fear not, that's an easy mistake to make — and I should know, because that's what I thought!

But the answer is no, this function is not tail-recursive. Although `sum(tail)` is at the end of the second `case` expression, you have to think like a compiler here, and when you do that you'll see that the last two actions of this function are:

1. Call `sum(xs)`
2. After that function call returns, add its value to `x` and return that result

When I make that code more explicit and write it as a series of one-line expressions, you see that it looks like this:

```
case x :: xs =>
    val s = sum(xs)
    val result = x + s
    return result
```

As shown, the *last calculation* that happens before the `return` statement is that the sum of `x` and `s` is calculated. If you're not 100% sure that you believe that, there are a few ways you can prove it to yourself.

1) Proving it with the previous "stack trace" example

One way to "prove" that the `sum` algorithm is not tail-recursive is with the stack trace output from the previous lesson. As you'll recall, the JVM output shows the `sum` method is called once for each step in the recursion — five times when the list contains five elements — so it's clear that the JVM feels the need to create a new instance of `sum` for each element in the collection.

2) Proving it with the `@tailrec` *annotation*

A second way to prove that `sum` isn't tail-recursive is to attempt to tag the function with a Scala annotation named `@tailrec`. This annotation is cool because your function won't compile unless it's tail-recursive.

For example, if you attempt to add the `@tailrec` annotation to `sum`, like this:

```
// need to import tailrec before using it
import scala.annotation.tailrec

@tailrec
```

```
def sum(list: List[Int]): Int = list match
    case Nil => 0
    case x :: xs => x + sum(xs)
```

the `scalac` compiler (or your IDE) will show an error message like this:

```
-- Error: ----------------------------------------------------
4 |    case x :: xs => x + sum(xs)
  |                         ^^^^^^^
  |                 Cannot rewrite recursive call: it is not
                    in tail position
1 error found
```

This is another way to "prove" that the Scala compiler doesn't think `sum` is tail-recursive.

So, how do I write a tail-recursive function?

Now that you know the current approach isn't tail-recursive, the question becomes, "How do I make it tail-recursive?"

A common pattern used to make a recursive function that "accumulates a result" into a tail-recursive function is to follow this series of steps:

1. Keep the original function signature the same (i.e., `sum`'s signature).
2. Create a second function by (a) copying the original function, (b) giving it a new name, (c) making it `private`, (d) giving it a new `accumulator` input parameter, and (e) adding the `@tailrec` annotation to it.
3. Modify the second function's algorithm so it uses the new accumulator. (More on this shortly.)
4. Call the second function from inside the first function. When you do this, you give the second function's `accumulator` parameter a

"seed" value, such as the *identity* value I wrote about in the previous lessons.

Let's jump into an example to see how this works.

Example: How to make sum tail-recursive

1) Keep the original function signature the same

To begin the process of converting the recursive `sum` function into a *tail-recursive* `sum` algorithm, leave the external signature of `sum` the same as it was before:

```
def sum(list: List[Int]): Int = ...
```

2) Create a second function

Now create the second function by copying the first function, giving it a new name, marking it `private`, giving it a new "accumulator" parameter (named `acc` in this example), and adding the `@tailrec` annotation to it:

```
@tailrec
private def sumWithAccumulator(list: List[Int], acc: Int): Int =
    list match
        case Nil => 0
        case x :: xs => x + sum(xs)
```

This code won't compile as shown, so I'll fix that next.

> TIP: Another key thing to notice in this solution is that the data type for the accumulator (`Int`) is the same as the data type held in the `List` that we're iterating over.

3) Modify the second function's algorithm

The third step is to modify the algorithm of the newly-created function to use the `accumulator` parameter. The easiest way to explain this is to show the code for the solution, and then explain the changes. Here's the source code (where I renamed the `accumulator` parameter to `acc` so the code would fit in the book's width):

```
@tailrec
private def sumWithAccumulator(list: List[Int], acc: Int): Int =
    list match
        case Nil => acc
        case x :: xs => sumWithAccumulator(xs, acc + x)
```

Here's a description of how that code works:

- I marked it with `@tailrec` so the compiler can help me by verifying that my code truly is tail-recursive.
- `sumWithAccumulator` takes two parameters, `list: List[Int]`, and `accumulator: Int`.
- The first parameter is the same list that the `sum` function receives.
- The second parameter is new. It's the "accumulator" that I mentioned earlier.
- The inside of the `sumWithAccumulator` function looks similar. It uses the same `match/case` approach that the original `sum` method used.
- Rather than returning `0`, the first `case` statement returns the

`accumulator` value when the `Nil` pattern is matched.

- The second `case` expression IS tail-recursive. When this `case` is matched, it immediately calls `sumWithAccumulator`, passing in the `xs` (tail) portion of `list`. What's different here is that the second parameter is the sum of (a) the `accumulator`, and (b) the head of the current list, `x`.
- Where the original `sum` method passed itself the tail of `xs` and then later added that result to `x`, this new approach keeps track of the accumulator (total sum) value as each recursive call is made.

The result of this approach is that the "last action" of the `sumWithAccumulator` function is this call:

```
sumWithAccumulator(xs, accumulator + x)
```

Because this last action really is a call back to the same function, the JVM can optimize this code as Mr. Odersky described earlier.

4) Call the second function from the first function

The fourth step in the process is to modify the original function to call the new function. Here's the source code for the new version of `sum`:

```
def sum(list: List[Int]): Int = sumWithAccumulator(list, 0)
```

Here's a description of how it works:

- The `sum` function signature is the same as before. It accepts a `List[Int]` and returns an `Int` value.
- The body of `sum` is just a call to the `sumWithAccumulator` function. It passes the original `list` to that function, and also gives its `accumulator` parameter an initial seed value of `0`.

Note that this *seed value* is the same as the *identity* value I wrote about in the previous recursion lessons. In those lessons I noted:

- The identity value for a sum algorithm is `0`
- The identity value for a product algorithm is `1`
- The identity value for a string concatenation algorithm is `""`

A few notes about the sum function

Looking at `sum` again:

```
def sum(list: List[Int]): Int = sumWithAccumulator(list, 0)
```

a few key points about it are:

- The intention of the design is that other programmers will call `sum`. It's the "Public API" portion of the solution.
- It has the same function signature as the previous version of `sum`. The benefit of this is that other programmers won't have to provide the initial seed value. In fact, they won't know that the internal algorithm uses a seed value. All they'll see in their IDE or your Scaladoc is `sum`'s type signature:

  ```
  def sum(list: List[Int]): Int
  ```

A slightly better way to write `sum`

Tail-recursive algorithms that use accumulators are typically written in the manner shown, with one exception: Rather than mark the new accumulator function as `private`, most Scala/FP developers like to put that function *inside* the original function as a way to limit its scope.

When doing this, the thought process is, "Don't expose the

scope of `sumWithAccumulator` in any way, unless you want other functions to call it."

When you make this change, the final code looks like this:

```
/**
 * A tail-recursive solution with the accumulator function
 * enclosed inside the outer `sum` function.
 */
import scala.annotation.tailrec

def sum(list: List[Int]): Int =
    @tailrec
    def sumWithAccumulator(list: List[Int], currSum: Int): Int =
        list match
            case Nil => currSum
            case x :: xs => sumWithAccumulator(xs, currSum + x)
    end sumWithAccumulator
    sumWithAccumulator(list, 0)
end sum
```

Feel free to use either approach. (Don't tell anyone, but I prefer the first approach; I think it reads more easily.)

A note on variable names

As shown in the previous example, if you don't like the name `accumulator` for the new input parameter, it may help to see the function with a different parameter name. For a "sum" algorithm a name like `runningTotal` or `currentSum` may be more meaningful:

```
/**
 * A complete tail-recursive solution that shows a
 * different name for the accumulator parameter.
 */
```

```
import scala.annotation.tailrec

def sum(list: List[Int]): Int =
    sumWithAccumulator(list, 0)

@tailrec
def sumWithAccumulator(
    list: List[Int],
    runningTotal: Int   // the 'accumulator' parameter
): Int = list match
    case Nil => runningTotal
    case x :: xs => sumWithAccumulator(xs, runningTotal + x)
```

(Note that I changed the formatting of that function to fit the book's width.)

I encourage you to use whatever name makes sense to you. Personally I prefer `currentSum` for this algorithm, but you'll often hear this approach referred to as using an "accumulator," which is why I used that name first.

Proving that this solution is tail-recursive

To wrap up this discussion, let's take a few moments to prove that the compiler thinks this code is tail-recursive.

First proof

As you may be thinking, the first proof is already in the code. When you compile this code with the `@tailrec` annotation and the compiler doesn't complain, you know that the compiler believes the code is tail-recursive.

Second proof

If for some reason you don't believe the compiler, a second way to prove this is to add some debug code to the new `sum` function, just like we did in the previous lessons. Here's the source code for a full Scala 3 application that shows this approach:

```
//> using scala "3"
//  run me like this:
//     scala-cli SumTailRecursive.scala

import scala.annotation.tailrec

def sum(list: List[Int]): Int =
    sumWithAccumulator(list, 0)

@tailrec
def sumWithAccumulator(
    list: List[Int],
    runningTotal: Int   // the 'accumulator' parameter
): Int = list match
    case Nil =>
      val stackTraceAsArray = Thread.currentThread.getStackTrace
        stackTraceAsArray.foreach(println)
        runningTotal
    case x :: xs =>
        sumWithAccumulator(xs, runningTotal + x)

@main def SumTailRecursive =
    println(sum(List.range(1, 10_000)))
```

When I put that code in a file named `SumTailRecursive.scala` and then compile and run it with `scala-cli`, I see a lot of other output related to the compilation process, followed by the answer:

```
$ scala-cli SecondProof.scala
// ... other compiler output here ...
java.base/java.lang.Thread.getStackTrace(Thread.java:1610)
SecondProof$package$.sumWithAccumulator(SecondProof.scala:16)
SecondProof$package$.sum(SecondProof.scala:8)
SecondProof$package$.SumTailRecursive(SecondProof.scala:23)
SumTailRecursive.main(SecondProof.scala:22)
49995000
```

As you can see, although the `List` in the code contains 10,000 elements, there's only one call to `sum`, and more importantly in this case, only one call to `sumAccumulator`. You can now safely call `sum` with even more elements and it will work just fine without blowing the stack. (Go ahead and test it!)

> NOTE: The upper limit of a Scala `Int` is `2,147,483,647`, so at some point you'll create a number that's too large for that. Fortunately a `Long` goes to `2^63-1` (which is `9,223,372,036,854,775,807`), so that problem is easily remedied. (If that's not big enough, use a `BigInt`.)

Key points

In this lesson I:

- Showed why the `sum` function I created in the previous lessons isn't tail-recursive
- Defined tail recursion
- Introduced the `@tailrec` annotation
- Showed how to write a tail-recursive function
- Showed a formula you can use to convert a simple recursive function to a tail-recursive function
- Showed ways to prove to yourself that a function is tail-recursive

> TIP: Personally, I'm usually not smart enough to write a tail-recursive function right away, so I usually write my algorithms using simple recursion, and then convert them to use tail-recursion.

13

Bonus: Processing I/O with Recursion

> To demonstrate something completely different with recursion, I've adapted the following lesson from my book, Learn Functional Programming Without Fear[a].
>
> Because most of this text comes from that book, you'll see that I occasionally refer to the code we write as being like *algebra*. In FP, that's the exact mindset: When we write FP code, it's just like we are mathematicians, and every pure function we write is like an equation, and then we combine those equations together in a series of expressions. The use of pure functions, immutable data, and immutable variables (*algebraic* variables) is what makes our code like algebra.
>
> [a] https://alvinalexander.com/scala/learn-functional-programming-book

This lesson's goal

Now that you've seen a *lot* of lessons that show how to use recursion to iterate over the elements in a `List`, let's look at another use of recursion: looping over a data source that is *not* a `List`.

In this lesson you'll write some recursive code to:

1. Prompt a user for their input
2. Read that input
3. Process the input as desired
4. Go back to Step 1

By the end of the lesson you'll see how to write recursive code to create

a complete, little command-line application that works like this:

```
$ scala-cli MainLoopExample.scala

Enter your name: al
AL
Enter your name: alvin
ALVIN
Enter your name: q
Q
```

To do this, I'm going to combine recursion along with (a) a `for` expression and (b) the Scala `Try` data type, so I'll briefly review those before we get into the recursion.

'for' expressions

A Scala `for` expression is like a `for` loop in other programming languages, but it *always yields* a result.

A `for` expression begins with the `for` keyword, iterates over a data source (or stream), lets you process that data as desired, and then ends by yielding a result. It has the following general syntax, including the `for` and `yield` keywords:

```
val result =
    for
        x <- xs   // 'xs' is a source of data
        y <- customFunction1(x)
        z <- customFunction2(y)
    yield
        // add business logic here as desired
        z
```

To demonstrate this more, here's a concrete example that shows how to

use the list `xs` as the data source for a `for` expression. This expression yields a new list `ys`, where each integer in `ys` is twice the value of the corresponding element in `xs`:

```
val xs = List(1, 2, 3)
val ys: List[Int] =
    for
        x <- xs
    yield
        x * 2

// ys has this value:
// ys == List(2, 4, 6)
```

I wrote that `for` expression in a long form to clearly show the `for/yield` sections, but you can also write it as a one-liner like this:

```
val ys = for x <- xs yield x * 2
```

In summary, a `for` expression is used to (a) loop over a list (or other data source), (b) process that data as desired, and (c) yield some result.

> TIP: If you're familiar with the `map` method on Scala collections classes, the `for` expression just shown works the same as this `map` method:
>
> ```
> val ys = xs.map(x => x * 2)
> ```

Error handling with Try

In addition to `for` expressions, the other piece of knowledge you need to know for the recursion example that follows is how to use Scala's `Try` data type.

A key thing to know is that `Try` is an *error-handling data type*. This means

that rather than writing a function that throws an exception, you return one of `Try`'s two sub-types, `Success` and `Failure`:

- When the function's algorithm *succeeds* as desired, the function returns its successful result inside a `Success`.
- When the function's algorithm *throws an exception*, the function catches that exception and then returns a `Failure` that contains the exception information.

If you're familiar with Java's `Optional` data type, `Try` is similar to that, but a significant difference is that `Try` gives us access to the exception information. Because of that, I use it all the time for I/O functions and any other function that can throw an exception. (That way I can tell the end user what went wrong.)

A Try example

To demonstrate `Try`, imagine that you want to write a pure function that converts a `String` to an `Int`. Because this function can receive bad input like `"foo"` or `"yo"` as well as good input like `"1"` or `"2"`, a pure function must account for that bad input.

Furthermore, because pure functions never throw exceptions — algebraic code *never* short-circuits with an exception — we commonly use `Try` in this situation:

```
// you have to import the Try data types to use them
import scala.util.{Try, Success, Failure}

// return Try instead of throwing an exception
def makeInt(s: String): Try[Int] =
    try
        Success(s.toInt)
```

```
    catch
        case e: NumberFormatException => Failure(e)
```

This is what `makeInt` looks like when it's called with both good and bad data:

```
// success case
makeInt("1")      // Success(1)

// failure case
makeInt("one")    // Failure(java.lang.NumberFormatException:
                  //         For input string: "one")
```

The way `makeInt` works is like this:

1. It receives a `String` input parameter, which I have named `s`
2. It attempts to convert that `String` to an `Int` by calling `s.toInt`
3. When that attempt succeeds, the result is wrapped in a `Success` value, and that `Success` is returned
4. Conversely, if that attempt fails, control is transferred to the `catch` block, and a `NumberFormatException` is returned inside a `Failure`

As mentioned, `Success` and `Failure` are sub-types of `Try`, so I declare the function to return the parent `Try` type, and then I return specific instances of `Success` and `Failure` inside the function.

> TIP 1: When you declare that a function returns a `Try[Int]`, this means that the `Success` value must contain an `Int` (while the `Failure` always contains an exception). And, as described, a function either returns a `Success` *or* a `Failure`.
>
> TIP 2: Once you have a `Try` value — such as a result from `makeInt` — a typical way to handle it is with a `match` expression:

```
makeInt(aString) match
    case Success(i) =>
         println(s"Success: i = $i")
    case Failure(e) =>
         println(s"Failed: msg = ${e.getMessage}")
```

A more concise version of Try

As a last point about `Try`, in a "basic" scenario like `makeInt` where you have (a) an algorithm that throws an exception, and (b) you don't want to do anything special inside the `try` and `catch` blocks, you can write your code more concisely like this:

```
def makeInt(s: String): Try[Int] = Try(s.toInt)
```

In this situation, `Try`'s constructor works just like the longer code previously shown.

Using recursion to create a loop

Given those two pieces of knowledge as background information, let's look at how to use recursion differently. In the rest of this lesson we'll use a `for` expression and recursion to stay in the following loop until the user says they want to exit:

1. Prompt a user for their input
2. Read that input
3. Process the input as desired
4. Go back to Step 1

Before we start on the "looping" part of the code, we'll first need two functions, one to prompt a user for their input, and a second to read

their input:

```
import scala.io.StdIn
import scala.util.{Try, Success, Failure}

def printOutput(s: String): Try[Unit] = Try {
    print(s)
}

def readInput(): Try[String] = Try {
    StdIn.readLine()
}
```

I use `Try` with these functions because I want you to imagine that these functions are REST API calls, where one writes to a REST endpoint and the other reads a REST response. Because network calls can fail, you need to handle that possibility, and `Try` is perfect for this.

Next, because I know that I want to continuously prompt a user and then read their input, I know that I want some sort of loop. The way I'm going to solve this is with the use of a `for` expression.

There's a little bit of a "chicken and the egg" thing going on here, so what I'm going to do is demonstrate a solution I know based on past experience, and then I'll explain that as I go on. Therefore, if you'll bear with me for a few moments, I'm going to start writing this code to create a "main loop":

```
def mainLoop(): Try[Unit] = for ...
```

Based on past experience, what I'm thinking here is this:

- I need to create a loop
- A `for` expression is a great "looping" tool
- Inside `for` I'm going to prompt the user, then read and process

their input

- One part you'll see as we go along is that immediately after this, I will then recursively call `mainLoop` from inside the `for` expression when the user wants to continue

Prompting for input

Given that background, let's focus on writing the code inside the `for` expression. In here we know that the first thing we want to do is prompt the user for their input, so I add this line:

```
def mainLoop(): Try[Unit] = for
    _ <- printOutput("Enter your name: ")
    // more code soon ...
```

This code is a little different than the `for` expression I showed earlier, but it can be read as, "Prompt the user for their input. Because `printOutput` returns a `Unit` value wrapped inside a `Try`, I don't care about that value, so ignore it." The key here is that I use the _ character on the left side of the `<-` operator to say, "I don't care about the value returned by `printOutput`."

I *could* also write that code with a variable name, like this:

```
def mainLoop(): Try[Unit] = for
    ignoreMe <- printOutput("Enter your name: ")
    // more code soon ...
```

but once you get used to seeing the _ character, it jumps out at you and makes the intention to "ignore this value" very obvious.

Reading the input

The next thing I want to do inside the loop is read the user's input, so I add in a `readInput` call:

```
def mainLoop(): Try[Unit] = for
    _     <- printOutput("Enter your name: ")
    input <- readInput()
    // more code soon ...
```

This new line of code reads the user input and binds it to the variable named `input`. What happens here is that whatever the user types in before they press the `[Enter]` key becomes a `String` that's assigned to the `input` variable.

A key to know here is that even though `readInput` returns a `Try[String]`, the variable `input` is a plain `String`. *How* this works is beyond the scope of this particular book, but this is the way that types like `Try` work: inside a `for` expression, you know that you're working with the value contained inside the `Success` type.

> Conversely, if `readInput` or `printOutput` return a `Failure`, the `for` expression is terminated at that point, and the `for` expression's result is a `Failure`. In this situation, that `Failure` would be `mainLoop`'s return type.

For the purposes of this discussion I don't care about potential errors, so we'll press on.

> For MANY more details about how `for` expressions work, see my book, Functional Programming, Simplified[a].
>
> [a] https://alvinalexander.com/scala/functional-programming-simplified-book

Handling the input

Next, our application needs to *process* the user's input. In a larger application you'd write a function to do this, but to keep things simple I'm just going to convert the user's input to uppercase. Therefore, I know that I want to do *something* like this:

```
def mainLoop(): Try[Unit] = for
    _       <- printOutput("Enter your name: ")
    input   <- readInput()
    _       <- {
                   val ucInput = input.toUpperCase
                   printOutput(ucInput + "\n")
                   // hmm, now i need to do something to
                   // loop again ...
               }
yield
    ()
```

A few notes about that code:

- Code inside curly braces is a "block of code (BOC)" (for lack of a better term)
- The value of a BOC is the value of the last expression inside the BOC
- Therefore, the type of this BOC is the return type of `printOutput`, which we know is `Try[Unit]`

Another important note is that the `yield` portion of the code returns the symbol `()`. This is the Scala way to say that this code yields an instance of the `Unit` type. `Unit` is like `void` or `Void` in other languages, and when you return it from a `yield`, it means that this expression does not return anything (or at least nothing of interest). Because I knew this was coming, that's how I knew `mainLoop`'s return type would be `Try[Unit]`:

```
def mainLoop(): Try[Unit] = for ...
                ---------
```

At this point, `mainLoop` works as-is and will prompt a user *one time*, but since I want to prompt them over and over again, we need to add something else …

Implementing the loop

When you're working with immutable values, the solution to this problem is almost always the same: recursion. Therefore, you may not know it yet, but what we need to do inside the `for` expression is to call `mainLoop` recursively, so it starts the prompt/read/handleInput process all over again.

Because I'm about to recursively call `mainLoop`, a good thing to do at this point is to think about the recursion's *end condition*. To keep things as simple as possible, my recursion will end when the uppercase version of the string I get from the user is a `"Q"`. When this happens, I'll stop the recursion by exiting the application.

I can sketch those thoughts as code like this:

```
if
   input.toUpperCase == "Q" then System.exit(0)
else
   mainLoop()   // the recursive call
```

With a few minor adjustments, I then add that code to the existing `for` expression:

```
import scala.util.{Try, Success, Failure}

def mainLoop(): Try[Unit] = for
    _       <- printOutput("Enter your name: ")
```

```
    input   <- readInput()
    _       <- {
                    val ucInput = input.toUpperCase
                    printOutput(ucInput + "\n")
                    if ucInput == "Q" then System.exit(0)
                    mainLoop()
                }
yield
    ()
```

You can format that code in different ways, but the important thing is that this code:

```
    _       <- {
                    val ucInput = input.toUpperCase
                    printOutput(ucInput + "\n")
                    if ucInput == "Q" then System.exit(0)
                    mainLoop()
                }
```

can be read as:

- As mentioned before, everything inside the curly braces is one block of code
- Inside that block, first convert the user's input to uppercase, and then print that value (`ucInput`)
- Next, if the uppercase version of their input is the string `"Q"`, exit the application
- Otherwise, call `mainLoop` recursively

Therefore, when you *don't* get `"Q"` as your input, `mainLoop` is called again, it starts its `for` expression, which prompts the user (again) for their input.

A common pattern

The approach I just showed is a very common way in FP to iterate over some source of data, whether that data is a `List`, user input, or any other data stream:

```
def mainLoop(): Try[Unit] = for
    data <- getData()
    _    <- {
                // [1] an algorithm that uses 'data'
                // [2] 'if' some condition is true,
                //     exit the loop, else:
                mainLoop()
            }
yield
    ()
```

As time goes on I'm sure that developers will find other ways to encapsulate this pattern (and some may exist already that I'm not aware of), but until then, I wanted to let you know that in FP this is currently a very common approach.

> TIP: As I write in *Functional Programming, Simplified* and *Learn Functional Programming Without Fear*, FPers write code just like a mathematician writes algebra: there are no moving parts, which means that in our code there are no mutable variables (`var` fields) and no mutable data structures (lists, etc.). Everything we write is truly like algebra. Because of this, `for` expressions and recursion are what we use to iterate over lists and streams.

A complete application

Lastly, I'll create a complete Scala 3 application to show that this works. The only new thing I'll add here is a Scala 3 `main` method, which is what kicks off a Scala 3 application. As you'll see in the code, all that `main` method does is make an initial call to `mainLoop` to get the ball rolling:

```
import scala.io.StdIn
import scala.util.{Try, Success, Failure}

// i shortened these two functions
def printOutput(s: String): Try[Unit] = Try(print(s))

def readInput(): Try[String] = Try(StdIn.readLine())

def mainLoop(): Try[Unit] = for
    _       <- printOutput("Enter your name: ")
    input   <- readInput()
    _       <- {
                    val ucInput = input.toUpperCase
                    printOutput(ucInput + "\n")
                    if ucInput == "Q" then System.exit(0)
                    mainLoop()
               }
yield
    ()

@main
def MainLoopExample =
    // this starts the application running:
    mainLoop()
```

The way this code works is:

- The Scala 3 compiler sees `@main` before the `MainLoopExample` method, and recognizes it as a main method

- A *main method* is the entry point to a Scala 3 application, so this method begins running
- It calls the `mainLoop` function, which that starts the process of prompting the user and reading their input

Lastly, as I showed in the beginning of this lesson, an interactive session with this application looks like this:

```
$ scala-cli MainLoopExample.scala

Enter your name: al
AL
Enter your name: alvin
ALVIN
Enter your name: q
Q
```

Key points

This was a relatively large lesson, so let's recap the key points:

- `for` expressions can be used to loop over lists, as well as streams of information (such as user input, in this example)
- The `Try` data type is used for error-handling, and it has the sub-types `Success` and `Failure`
- As shown in the `mainLoop` example, you can use recursion inside a `for` expression

I included this example in this booklet because, again, whenever you're using only immutable values, immutable data, and you also need to loop (or iterate) over that data, recursion is *the* solution. As shown in this example, that applies to processing user input, but if you imagine that `readInput` and `printOutput` were REST functions that interact with data across the internet, the solution would be the same.

One thing I glossed over in this lesson is that `Try` can be used inside `for` expressions. I didn't get into that here because it's a long story, but a very short answer is that `Try` works because it implements `map` and `flatMap` methods, and *any* class that properly implements those methods can work in `for` expressions. For much more detail on this, see my "Big FP Book," Functional Programming, Simplified[1].

[1] https://alvinalexander.com/scala/functional-programming-simplified-book

14

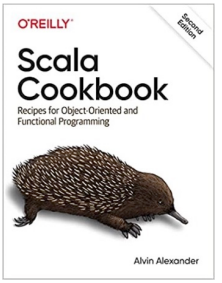
The End

As I mentioned way back in the beginning of this booklet, I pulled most of these chapters from my book, Functional Programming, Simplified: Updated for Scala 3[1].

In these chapters, because I attempt to explain recursion in a variety of different ways over about 100 pages, I thought this might make a nice, standalone booklet. So if you're interested in recursion, I hope this booklet has been helpful.

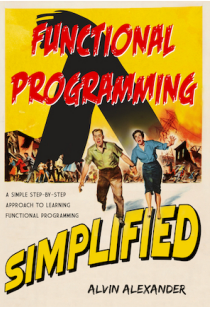
Other books

If you're interested in other books I've written on Scala, here's the current list as of January, 2023:

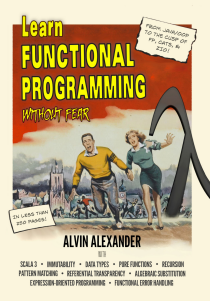


Scala Cookbook, 2nd Edition (Amazon.com)[2]

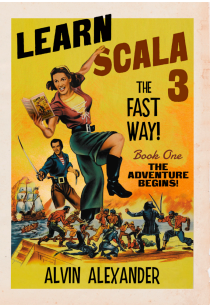
[1] https://alvinalexander.com/scala/functional-programming-simplified-book-scala-3
[2] https://amzn.to/3du1pMR



Functional Programming, Simplified
("The Big FP Book," alvinalexander.com)[3]



Learn Functional Programming Without Fear
("The Little FP Book," alvinalexander.com)[4]



Learn Scala 3 The Fast Way![5]

[3] https://alvinalexander.com/scala/functional-programming-simplified-book
[4] https://alvinalexander.com//scala/learn-functional-programming-book
[5] https://alvinalexander.com/scala/learn-scala-3-the-fast-way-book

Other resources

If you're interested in other free resources related to recursion, here's a small collection of blog posts I've written:

- A Scala 'foldLeft' function written using recursion[6]
- More Scala recursion examples[7]
- A Scala factorial recursion example[8]
- Recursion is great, but check out Scala's fold and reduce[9]
- A quick review of Scala's for-expressions[10]
- You can also find my free videos here on my YouTube channel[11]

Support my writing

As a last note, if you'd like to see more free documents like this in the future (and free videos), you can support my work here:

- "Buy me a coffee" at ko-fi.com/alvin[12]
- Be a patron of my work at patreon.com/alvinalexander[13]

[6] https://alvinalexander.com/source-code/scala-foldleft-function-using-recursion/
[7] https://alvinalexander.com/scala/scala-recursion-examples-recursive-programming
[8] https://alvinalexander.com/scala/scala-factorial-recursion-example-recursive-programming/
[9] https://alvinalexander.com/scala/fp-book/recursion-great-but-fold-reduce-in-scala/
[10] https://alvinalexander.com/scala/fp-book/quick-review-scala-for-expressions/
[11] https://www.youtube.com/@devdaily/videos
[12] https://ko-fi.com/alvin
[13] https://www.patreon.com/alvinalexander

Find me here

You can also find me at these locations:

- alvinalexander.com[14]
- twitter.com/alvinalexander[15]
- linkedin.com/in/alvinalexander[16]

All the best,
Alvin Alexander
January, 2023

[14] https://alvinalexander.com
[15] https://twitter.com/alvinalexander
[16] https://www.linkedin.com/in/alvinalexander

Index