Toggle navigation Scala Exercises

- Login with GitHub
- Terms And Types
- Definitions And Evaluation
- Functional Loops
- Lexical Scopes
- Tail Recursion
- Structuring Information
- Higher Order Functions
- Standard Library
- Syntactic Conveniences
- Object Oriented Programming
- Imperative Programming
- Classes Vs Case Classes
- Polymorphic Types
- Lazy Evaluation
- Type Classes

Scala Tutorial

0 / 15

- **Sections**

View on GitHub

Tail Recursion

Recursive Function Application

Let’s compare the evaluation steps of the application of two recursive methods.

First, consider `gcd`, a method that computes the greatest common divisor of two numbers.

Here's an implementation of `gcd` using Euclid's algorithm.

```
def gcd(a: Int, b: Int): Int =
  if (b == 0) a else gcd(b, a % b)
```

OPEN CHAT

`gcd(14, 21)` is evaluated as follows:

```
gcd(14, 21)
if (21 == 0) 14 else gcd(21, 14 % 21)
if (false) 14 else gcd(21, 14 % 21)
gcd(21, 14 % 21)
gcd(21, 14)
if (14 == 0) 21 else gcd(14, 21 % 14)
if (false) 21 else gcd(14, 21 % 14)
gcd(14, 7)
gcd(7, 14 % 7)
gcd(7, 0)
if (0 == 0) 7 else gcd(0, 7 % 0)
if (true) 7 else gcd(0, 7 % 0)
7
```

Now, consider `factorial`:

```
def factorial(n: Int): Int =
  if (n == 0) 1 else n * factorial(n - 1)
```

`factorial(4)` is evaluated as follows:

```
factorial(4)
if (4 == 0) 1 else 4 * factorial(4 - 1)
4 * factorial(3)
4 * (3 * factorial(2))
4 * (3 * (2 * factorial(1)))
4 * (3 * (2 * (1 * factorial(0)))
4 * (3 * (2 * (1 * 1)))
24
```

What are the differences between the two sequences?

One important difference is that in the case of `gcd`, we see that the reduction sequence essentially oscillates. It goes from one call to `gcd` to the next one, and eventually it terminates. In between we have expressions that are different from a simple call like if then else's but we always come back to this shape of the call of `gcd`. If we look at `factorial`, on the other hand we see that in each couple of steps we add one more element to our expressions. Our expressions becomes bigger and bigger until we end when we finally reduce it to the final value.

Tail Recursion

That difference in the rewriting rules actually translates directly to a difference in the actual execution on a computer. In fact, it turns out that if you have a recursive function that calls itself as its last action, then you can reuse the stack frame of that function. This is called *tail recursion*.

And by applying that trick, a tail recursive function can execute in constant stack space, so it's really just another formulation of an iterative process. We could say a tail recursive function is the functional form of a loop, and it executes just as efficiently as a loop.

If we look back at `gcd`, we see that in the else part, `gcd` calls itself as its last action. And that translates to a rewriting sequence that was essentially constant in size, and that will, in the actual execution on a computer, translate into a tail recursive call that can execute in constant space.

On the other hand, if you look at `factorial` again, then you'll see that after the call to `factorial(n - 1)`, there is still work to be done, namely, we had to multiply the result of that call with the number `n`. So, that recursive call is not a tail recursive call, and it becomes evident in the reduction sequence, where you see that actually there’s a buildup of intermediate results that we all have to keep until we can compute the final value. So, `factorial` would not be a tail recursive function.

Both `factorial` and `gcd` only call itself but in general, of course, a function could call other functions. [illegible] of tail recursion is that, if the last action of a function consists of calling another function, maybe the other function, the stack frame could be reused for both functions. Such calls are called *tail calls*.

Tail Recursion in Scala

In Scala, only directly recursive calls to the current function are optimized.

One can require that a function is tail-recursive using a `@tailrec` annotation:

```
@tailrec
def gcd(a: Int, b: Int): Int = …
```

If the annotation is given, and the implementation of `gcd` were not tail recursive, an error would be issued.

Exercise

Complete the following definition of a tail-recursive version of `factorial`:

▶ Run

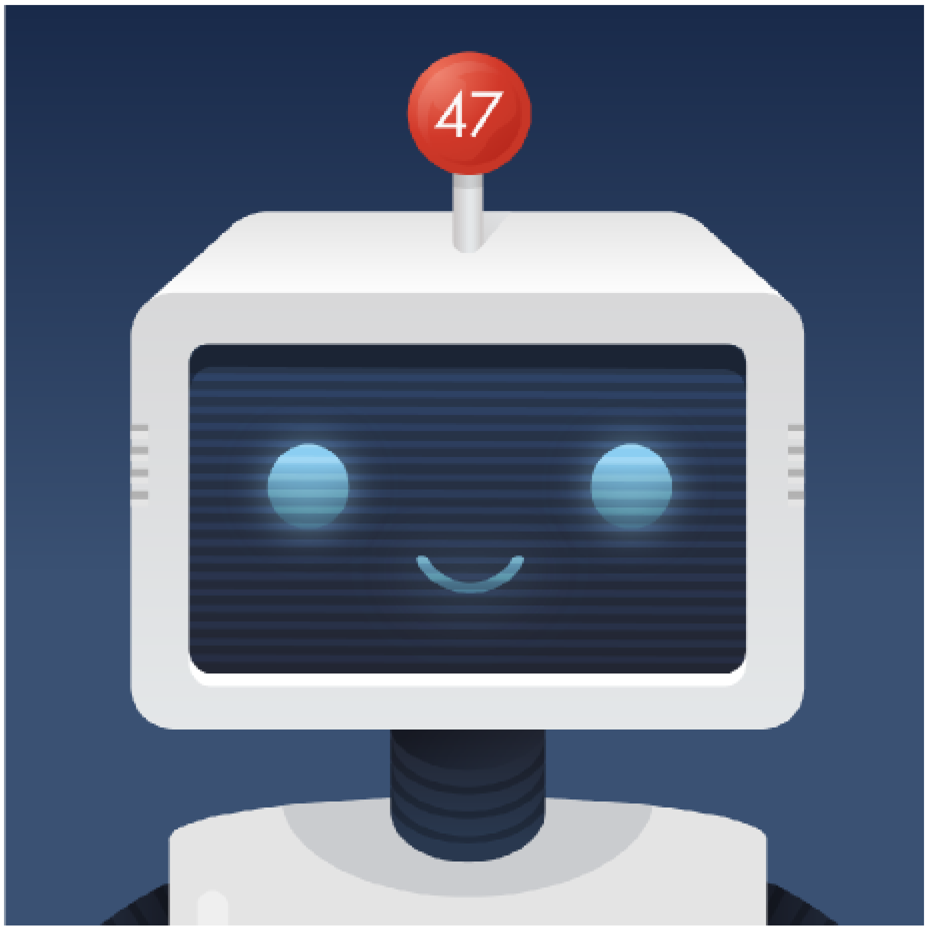
```
def factorial(n: Int): Int = {
  @tailrec
  def iter(x: Int, result: Int): Int =
    if (x ==                    ) result
    else iter(x - 1, result * x)

  iter(n,                    )
}

factorial(3) shouldBe 6
factorial(4) shouldBe 24
```

- 6 contributors
-

OPEN CHAT

-
- 

OPEN CHAT

-
-

OPEN CHAT

-
- Edit exercises

Scala Exercises is an Open Source project by 47 Degrees

- Contributing
- ·
- License

×

Users who have contributed to this file

-

  Update scalafmt-core to 2.6.1 (#135)

  BenFradet contributed 2020-06-24T12:47:39Z

OPEN CHAT

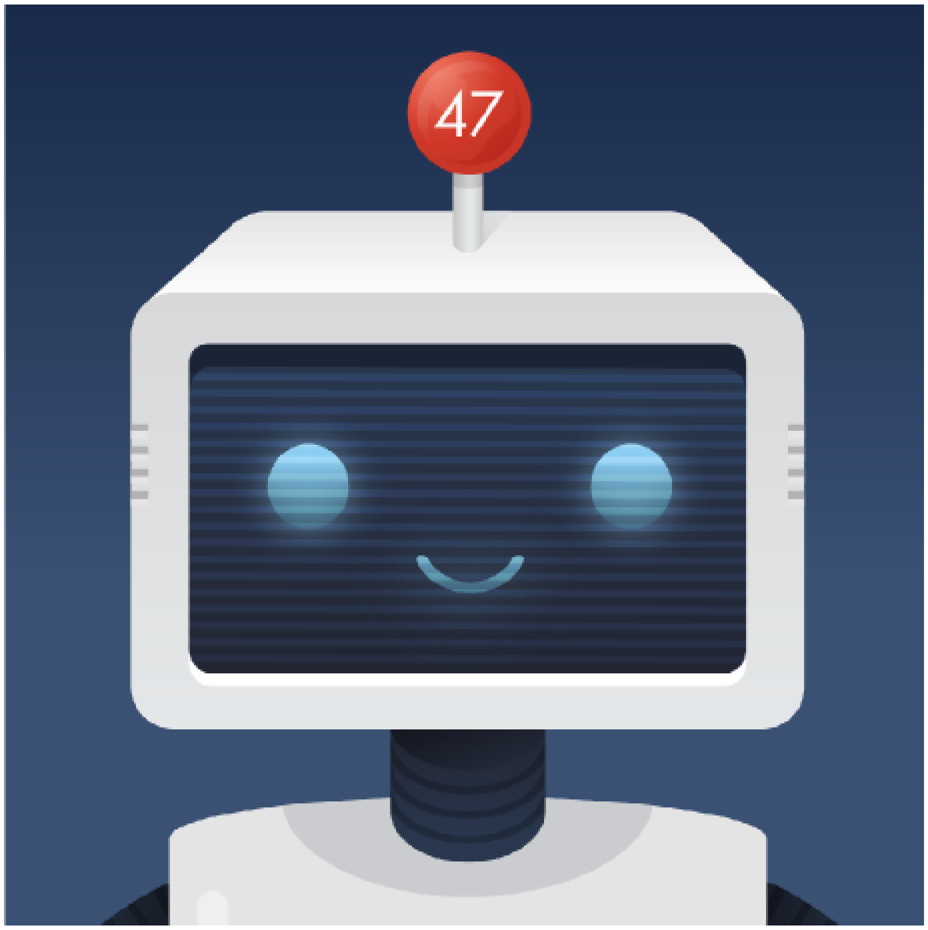
View on GitHub

-

  Prepare repository for next release and SBT build improvements (#128)

  juanpedromoreno contributed 2020-06-18T14:39:02Z
  View on GitHub

OPEN CHAT

- 

  Update documentation and other files (#117) Co-authored-by: github-actions[bot] <41898282+github-actions[bot]@users.noreply.github.com>

  47erbot contributed 2020-04-24T08:02:44Z
  View on GitHub

OPEN CHAT

-

  Eliminated possibilities to perform infinite loops

  kiroco12 contributed 2020-01-31T11:25:31Z
  View on GitHub
-

OPEN CHAT

Updated Scala version and dependencies

kiroco12 contributed 2019-11-20T08:51:31Z
View on GitHub

-

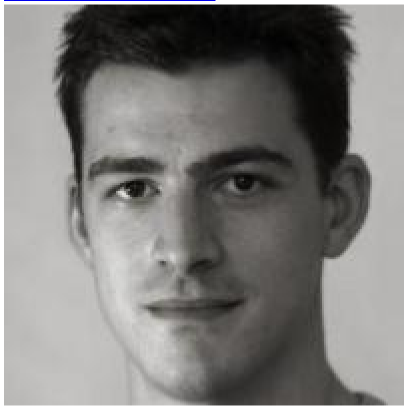
  Update TailRecursion.scala

  timaschew contributed 2017-09-01T12:15:09Z
  View on GitHub

OPEN CHAT

-

  Integrates sbt-org-policies plugin (#9)

  juanpedromoreno contributed 2017-03-28T18:17:03Z
  View on GitHub
- 

  Add TailRecursion section (end of week1)

  julienrf contributed 2016-11-16T14:02:43Z
  View on GitHub

OPEN CHAT

-

  Add tutorial structure. Add the first two sections.

  julienrf contributed 2016-11-15T17:00:02Z
  View on GitHub

×

Sign up

Login with GitHub

OPEN CHAT