Functional Programming Principles in Scala

Principles of Functional Programming

Martin Odersky

Programming Paradigms

Paradigm: In science, a *paradigm* describes distinct concepts or thought patterns in some scientific discipline.

Main programming paradigms:

- imperative programming
- functional programming
- logic programming

Orthogonal to it:

- object-oriented programming

Review: Imperative programming

Imperative programming is about

- modifying mutable variables,
- using assignments
- and control structures such as if-then-else, loops, break, continue, return.

The most common informal way to understand imperative programs is as instruction sequences for a Von Neumann computer.

Imperative Programs and Computers

There's a strong correspondence between

Mutable variables	$\approx$	memory cells
Variable dereferences	$\approx$	load instructions
Variable assignments	$\approx$	store instructions
Control structures	$\approx$	jumps

Problem: Scaling up. How can we avoid conceptualizing programs word by word?

Reference: John Backus, Can Programming Be Liberated from the von. Neumann Style?, Turing Award Lecture 1978.

Scaling Up

In the end, pure imperative programming is limited by the "Von Neumann" bottleneck:

> *One tends to conceptualize data structures word-by-word.*

We need other techniques for defining high-level abstractions such as collections, polynomials, geometric shapes, strings, documents.

Ideally: Develop *theories* of collections, shapes, strings, ...

What is a Theory?

A theory consists of

- one or more *data types*
- *operations* on these types
- *laws* that describe the relationships between values and operations

Normally, a theory does not describe mutations!

Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

```
(a*x + b) + (c*x + d) = (a + c)*x + (b + d)
```

But it does not define an operator to change a coefficient while keeping the polynomial the same!

Theories without Mutation

For instance the theory of polynomials defines the sum of two polynomials by laws such as:

```
(a*x + b) + (c*x + d) = (a + c)*x + (b + d)
```

But it does not define an operator to change a coefficient while keeping the polynomial the same!

Whereas in an imperative program one *can* write:

```
class Polynomial { double[] coefficient; }
Polynomial p = ...;
p.coefficient[0] = 42;
```

Theories without Mutation

Other example:

The theory of strings defines a concatenation operator $++$ which is associative:

```
(a ++ b) ++ c  =  a ++ (b ++ c)
```

But it does not define an operator to change a sequence element while keeping the sequence the same!

(This one, some languages *do* get right; e.g. Java's strings are immutable)

Consequences for Programming

If we want to implement high-level concepts following their mathematical theories, there's no place for mutation.

- The theories do not admit it.
- Mutation can destroy useful laws in the theories.

Therefore, let's

- concentrate on defining theories for operators expressed as functions,
- avoid mutations,
- have powerful ways to abstract and compose functions.

Functional Programming

- In a *restricted* sense, functional programming (FP) means programming without mutable variables, assignments, loops, and other imperative control structures.
- In a *wider* sense, functional programming means focusing on the functions and immutable data.
- In particular, functions can be values that are produced, consumed, and composed.
- All this becomes easier in a functional language.

Functional Programming Languages

- In a *restricted* sense, a functional programming language is one which does not have mutable variables, assignments, or imperative control structures.
- In a *wider* sense, a functional programming language enables the construction of elegant programs that focus on functions and immutable data structures.
- In particular, functions in a FP language are first-class citizens. This means
  - they can be defined anywhere, including inside other functions
  - like any other value, they can be passed as parameters to functions and returned as results
  - as for other values, there exists a set of operators to compose functions

Some functional programming languages

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell
- Scala

By now, concepts and constructs from functional languages are also found in many traditional languages.

History of FP languages

1959	(Lisp)	2003	Scala
1975-77	ML, FP, Scheme	2005	F#
1978	(Smalltalk)	2007	Clojure
1986	Standard ML	2017	Idris
1990	Haskell, Erlang	2020	Scala 3
2000	OCaml		

Scala 3 is the language we will use in this course.

Origins of FP

1930s: Lambda Calculus (Alonzo Church)

- Shown to be equivalent to Turning Machines
- Stays relevant today as one of the theoretical foundations of FP

1959: Lisp

- Functions and recursive data tools for artifical intelligence research

1980/90s: ML, Haskell, ...

- New type systems with a strong connection to mathematical logic

Why Functional Programming?

- Reduce errors
- Improve modularity
- Higher-level abstractions
- Shorter code
- Increased developer productivity

Why Functional Programming Now?

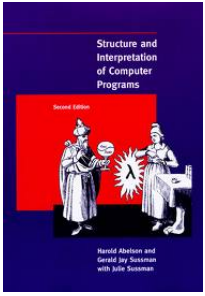
1. It's an effective tool to handle concurrency and parallelism, on every scale.
2. Our computers are not Van-Neuman machines anymore. They have
   - parallel cores
   - clusters of servers
   - distribution in the cloud

This causes new programming challenges such as

- cache coherency
- non-determinism

Recommended Book (1)

Structure and Interpretation of Computer Programs. Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.

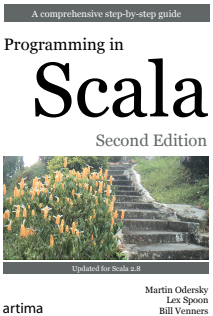


A classic. Many parts of the course and quizzes are based on it, but we change the language from Scheme to Scala.

The full text can be downloaded here.

Recommended Book (2)

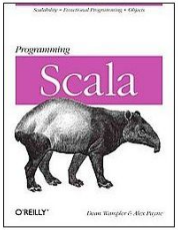
Programming in Scala. Martin Odersky, Lex Spoon, and Bill Venners. 4th edition. Artima 2019.

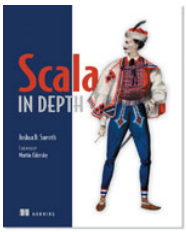


The standard language introduction and reference.

Other Recommended Books

There areb many other good introductions to Scala. Among them:





Elements of Programming

Principles of Functional Programming

Elements of Programming

Every non-trivial programming language provides:

- primitive expressions representing the simplest elements
- ways to *combine* expressions
- ways to *abstract* expressions, which introduce a name for an expression by which it can then be referred to.

The Read-Eval-Print Loop

Functional programming is a bit like using a calculator

An interactive shell (or REPL, for Read-Eval-Print-Loop) lets one write expressions and responds with their value.

The Scala REPL can be started by simply typing

```
> scala
```

Expressions

Here are some simple interactions with the REPL

```
scala> 87 + 145
res0: Int = 232
```

Functional programming languages are more than simple calcululators because they let one define values and functions:

```
scala> def size = 2
size: Int


scala> 5 * size
res1: Int = 10
```

Evaluation

A non-primitive expression is evaluated as follows.

1. Take the leftmost operator
2. Evaluate its operands (left before right)
3. Apply the operator to the operands

A name is evaluated by replacing it with the right hand side of its definition

The evaluation process stops once it results in a value

A value is a number (for the moment)

Later on we will consider also other kinds of values

Example

Here is the evaluation of an arithmetic expression:

```
def pi = 3.14159

def radius = 10

(2 * pi) * radius
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

```
6.28318 * radius
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

```
6.28318 * radius
```

```
6.28318 * 10
```

Example

Here is the evaluation of an arithmetic expression:

```
(2 * pi) * radius
```

```
(2 * 3.14159) * radius
```

```
6.28318 * radius
```

```
6.28318 * 10
```

```
62.8318
```

Parameters

Definitions can have parameters. For instance:

```
scala> def square(x: Double) = x * x
square: (x: Double)Double

scala> square(2)
4.0

scala> square(5 + 4)
81.0

scala> square(square(4))
256.0

scala> def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (x: Double, y: Double)Double
```

Parameter and Return Types

Function parameters come with their type, which is given after a colon

```
def power(x: Double, y: Int): Double = ...
```

If a return type is given, it follows the parameter list.

Primitive types are as in Java, but are written capitalized:

`Int`	32-bit integers
`Long`	64-bit integers
`Float`	32-bit floating point numbers
`Double`	64-bit floating point numbers
`Char`	16-bit unicode characters
`Short`	16-bit integers
`Byte`	8-bit integers
`Boolean`	boolean values `true` and `false`

Evaluation of Function Applications

Applications of parameterized functions are evaluated in a similar way as operators:

1. Evaluate all function arguments, from left to right
2. Replace the function application by the function's right-hand side, and, at the same time
3. Replace the formal parameters of the function by the actual arguments.

Example

```
sumOfSquares(3, 2+2)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
```

Example

```
sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9 + 4 * 4
9 + 16
25
```

The substitution model

This scheme of expression evaluation is called the *substitution model*.

The idea underlying this model is that all evaluation does is *reduce an expression to a value*.

It can be applied to all expressions, as long as they have no side effects.

The substitution model is formalized in the *λ-calculus*, which gives a foundation for functional programming.

Termination

- *Does every expression reduce to a value (in a finite number of steps)?*

Termination

- *Does every expression reduce to a value (in a finite number of steps)?*
- *No. Here is a counter-example*

```
def loop: Int = loop

loop
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4
```

Changing the evaluation strategy

The interpreter reduces function arguments to values before rewriting the function application.

One could alternatively apply the function to unreduced arguments.

For instance:

```
sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9 + 4 * (2+2)
9 + 4 * 4
25
```

Call-by-name and call-by-value

The first evaluation strategy is known as *call-by-value*, the second is is known as *call-by-name*.

Both strategies reduce to the same final values as long as

- the reduced expression consists of pure functions, and
- both evaluations terminate.

Call-by-value has the advantage that it evaluates every function argument only once.

Call-by-name has the advantage that a function argument is not evaluated if the corresponding parameter is unused in the evaluation of the function body.

Call-by-name vs call-by-value

Question: Say you are given the following function definition:

```
def test(x: Int, y: Int) = x * x
```

For each of the following function applications, indicate which evaluation strategy is fastest (has the fewest reduction steps)

CBV fastest	CBN fastest	same #steps	
0	0	0	`test(2, 3)`
0	0	0	`test(3+4, 8)`
0	0	0	`test(7, 2*4)`
0	0	0	`test(3+4, 2*4)`

Call-by-name vs call-by-value

```
def test(x: Int, y: Int) = x * x

test(2, 3)
test(3+4, 8)
test(7, 2*4)
test(3+4, 2*4)
```

EPFL

Evaluation Strategies and Termination

Principles of Functional Programming

Call-by-name, Call-by-value and termination

You know from the last module that the call-by-name and call-by-value evaluation strategies reduce an expression to the same value, as long as both evaluations terminate.

But what if termination is not guaranteed?

We have:

- If CBV evaluation of an expression e terminates, then CBN evaluation of e terminates, too.
- The other direction is not true

Non-termination example

Question: Find an expression that terminates under CBN but not under CBV.

Non-termination example

Let's define

```
def first(x: Int, y: Int) = x
```

and consider the expression `first(1, loop)`.

Under CBN:

```
first(1, loop)
```

Under CBV:

```
first(1, loop)
```

Scala's evaluation strategy

Scala normally uses call-by-value.

But if the type of a function parameter starts with => it uses call-by-name.

Example:

```
def constOne(x: Int, y: => Int) = 1
```

Let's trace the evaluations of

```
constOne(1+2, loop)
```

and

```
constOne(loop, 1+2)
```

Trace of `constOne(1 + 2, loop)`

```
constOne(1 + 2, loop)
```

Trace of `constOne(1 + 2, loop)`

```
constOne(1 + 2, loop)
constOne(3, loop)
```

Trace of `constOne(1 + 2, loop)`

```
constOne(1 + 2, loop)
constOne(3, loop)
1
```

Trace of `constOne(loop, 1 + 2)`

```
constOne(loop, 1 + 2)
```

Trace of `constOne(loop, 1 + 2)`

```
constOne(loop, 1 + 2)
constOne(loop, 1 + 2)
constOne(loop, 1 + 2)
...
```

Conditionals and Value Definitions

Principles of Functional Programming

Conditional Expressions

To express choosing between two alternatives, Scala has a conditional expression `if-then-else`.

It resembles an `if-else` in Java, but is used for expressions, not statements.

Example:

```
def abs(x: Int) = if x >= 0 then x else -x
```

`x >= 0` is a *predicate*, of type `Boolean`.

Boolean Expressions

Boolean expressions `b` can be composed of

```
true  false       // Constants
!b                // Negation
b && b            // Conjunction
b || b            // Disjunction
```

and of the usual comparison operations:

`e <= e`, `e >= e`, `e < e`, `e > e`, `e == e`, `e != e`

Rewrite rules for Booleans

Here are reduction rules for Boolean expressions (`e` is an arbitrary expression):

```
!true        -->   false
!false       -->   true
true && e    -->   e
false && e   -->   false
true || e    -->   true
false || e   -->   e
```

Note that `&&` and `||` do not always need their right operand to be evaluated.

We say, these expressions use “short-circuit evaluation”.

Exercise: Formulate rewrite rules for if-then-else

Value Definitions

We have seen that function parameters can be passed by value or be passed by name.

The same distinction applies to definitions.

The `def` form is "by-name", its right hand side is evaluated on each use.

There is also a `val` form, which is "by-value". Example:

```
val x = 2
val y = square(x)
```

The right-hand side of a `val` definition is evaluated at the point of the definition itself.

Afterwards, the name refers to the value.

For instance, `y` above refers to `4`, not `square(2)`.

Value Definitions and Termination

The difference between `val` and `def` becomes apparent when the right hand side does not terminate. Given

```
def loop: Boolean = loop
```

A definition

```
def x = loop
```

is OK, but a definition

```
val x = loop
```

will lead to an infinite loop.

Exercise

Write functions `and` and `or` such that for all argument expressions `x` and `y`:

```
and(x, y)    ==    x && y
or(x, y)     ==    x || y
```

(do not use `||` and `&&` in your implementation)

What are good operands to test that the equalities hold?

Example: Square roots with Newton's method

August 31, 2012

Task

We will define in this session a function

```
/** Calculates the square root of parameter x */
def sqrt(x: Double): Double = ...
```

The classical way to achieve this is by successive approximations using Newton's method.

Method

To compute `sqrt(x)`:

- Start with an initial *estimate* `y` (let's pick `y = 1`).
- Repeatedly improve the estimate by taking the mean of `y` and `x/y`.

Example: $x = 2$

Estimation	Quotient	Mean
1	2 / 1 = 2	1.5
1.5	2 / 1.5 = 1.333	1.4167
1.4167	2 / 1.4167 = 1.4118	1.4142
1.4142	...	...

Implementation in Scala (1)

First, define a function which computes one iteration step

```
def sqrtIter(guess: Double, x: Double): Double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)
```

Note that `sqrtIter` is *recursive*, its right-hand side calls itself.

Recursive functions need an explicit return type in Scala.

For non-recursive functions, the return type is optional

Implementation in Scala (2)

Second, define a function `improve` to improve an estimate and a test to check for terminatation:

```
def improve(guess: Double, x: Double) =
  (guess + x / guess) / 2

def isGoodEnough(guess: Double, x: Double) =
  abs(guess * guess - x) < 0.001
```

Implementation in Scala (3)

Third, define the `sqrt` function:

```
def sqrt(x: Double) = srqtIter(1.0, x)
```

Exercise

1. The `isGoodEnough` test is not very precise for small numbers and can lead to non-termination for very large numbers. Explain why.
2. Design a different version of `isGoodEnough` that does not have these problems.
3. Test your version with some very very small and large numbers, e.g.

```
0.001
0.1e-20
1.0e20
1.0e50
```

Blocks and Lexical Scope

Principles of Functional Programming

Nested functions

It's good functional programming style to split up a task into many small functions.

But the names of functions like `sqrtIter`, `improve`, and `isGoodEnough` matter only for the *implementation* of `sqrt`, not for its *usage*.

Normally we would not like users to access these functions directly.

We can achieve this and at the same time avoid "name-space pollution" by putting the auxciliary functions inside `sqrt`.

The sqrt Function, Take 2

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if isGoodEnough(guess, x) then guess
    else sqrtIter(improve(guess, x), x)

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0, x)
}
```

Blocks in Scala

- A block is delimited by braces `{ ... }`.

```
{ val x = f(3)
  x * x
}
```

- It contains a sequence of definitions or expressions.
- The last element of a block is an expression that defines its value.
- This return expression can be preceded by auxiliary definitions.
- Blocks are themselves expressions; a block may appear everywhere an expression can.
- In Scala 3, braces are optional (i.e. implied) around a correctly indented expression that appears after `=`, `then`, `else`, ...

Blocks and Visibility

```
val x = 0
def f(y: Int) = y + 1
val result =
  val x = f(3)
  x * x
```

- The definitions inside a block are only visible from within the block.
- The definitions inside a block *shadow* definitions of the same names outside the block.

Exercise: Scope Rules

Question: What is the value of `result` in the following program?

```
val x = 0
def f(y: Int) = y + 1
val y =
  val x = f(3)
  x * x
val result =  y + x
```

Possible answers:

- 0 `0`
- 0 `16`
- 0 `32`
- 0 `reduction does not terminate`

Exercise: Scope Rules

Question: What is the value of `result` in the following program?

```
val x = 0
def f(y: Int) = y + 1
val y =
  val x = f(3)
  x * x
val result =  y + x
```

Possible answers:

- 0 `0`
- 0 `16`
- 0 `32`
- 0 `reduction does not terminate`

Lexical Scoping

Definitions of outer blocks are visible inside a block unless they are shadowed.

Therefore, we can simplify `sqrt` by eliminating redundant occurrences of the `x` parameter, which means everywhere the same thing:

The sqrt Function, Take 3

```
def sqrt(x: Double) =
  def sqrtIter(guess: Double): Double =
    if isGoodEnough(guess) then guess
    else sqrtIter(improve(guess))

  def improve(guess: Double) =
    (guess + x / guess) / 2

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001

  sqrtIter(1.0)
```

Semicolons

If there are more than one statements on a line, they need to be separated by semicolons:

```
val y = x + 1; y * y
```

Semicolons at the end of lines are usually left out.

You could write

```
val x = 1;
```

but it would not be very idiomatic in Scala.

Summary

You have seen simple elements of functional programing in Scala.

- arithmetic and boolean expressions
- conditional expressions if-then-else
- functions with recursion
- nesting and lexical scope

You have learned the difference between the call-by-name and call-by-value evaluation strategies.

You have learned a way to reason about program execution: reduce expressions using the substitution model.

This model will be an important tool for the coming sessions.

Tail Recursion

Principles of Functional Programming

Review: Evaluating a Function Application

One simple rule : One evaluates a function application $\texttt{f}(e_1, ..., e_n)$

- by evaluating the expressions $e_1, \ldots, e_n$ resulting in the values $v_1, ..., v_n$, then
- by replacing the application with the body of the function `f`, in which
- the actual parameters $v_1, ..., v_n$ replace the formal parameters of `f`.

Application Rewriting Rule

This can be formalized as a *rewriting of the program itself*:

$$\begin{array}{l} \texttt{def f}(x_1, ..., x_n) = B;\ \ ...\ \ \texttt{f}(v_1, ..., v_n) \\ \rightarrow \\ \texttt{def f}(x_1, ..., x_n) = B;\ \ ...\ \ [v_1/x_1, ..., v_n/x_n]\, B \end{array}$$

Here, $[v_1/x_1, ..., v_n/x_n]\, B$ means:

The expression B in which all occurrences of x_i have been replaced by v_i.

$[v_1/x_1, ..., v_n/x_n]$ is called a *substitution*.

Rewriting example:

Consider `gcd`, the function 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 then a else gcd(b, a % b)
```

Rewriting example:

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

```
gcd(14, 21)
```

Rewriting example:

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

```
gcd(14, 21)
```

$\longrightarrow$ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

Rewriting example:

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

```
gcd(14, 21)
```

$\rightarrow$ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if false then 14 else gcd(21, 14 % 21)`

Rewriting example:

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

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
```

Rewriting example:

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

```
gcd(14, 21)
```

$\rightarrow$ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if false then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14)`

Rewriting example:

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

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
```

Rewriting example:

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

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
```

Rewriting example:

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

`gcd(14, 21)`

$\rightarrow$ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if false then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14)`

$\rightarrow$ `if 14 == 0 then 21 else gcd(14, 21 % 14)`

$\twoheadrightarrow$ `gcd(14, 7)`

$\twoheadrightarrow$ `gcd(7, 0)`

Rewriting example:

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

```
gcd(14, 21)
→ if 21 == 0 then 14 else gcd(21, 14 % 21)
→ if false then 14 else gcd(21, 14 % 21)
→ gcd(21, 14 % 21)
→ gcd(21, 14)
→ if 14 == 0 then 21 else gcd(14, 21 % 14)
↠ gcd(14, 7)
↠ gcd(7, 0)
→ if 0 == 0 then 7 else gcd(0, 7 % 0)
```

Rewriting example:

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

`gcd(14, 21)`

$\rightarrow$ `if 21 == 0 then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `if false then 14 else gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14 % 21)`

$\rightarrow$ `gcd(21, 14)`

$\rightarrow$ `if 14 == 0 then 21 else gcd(14, 21 % 14)`

$\twoheadrightarrow$ `gcd(14, 7)`

$\twoheadrightarrow$ `gcd(7, 0)`

$\rightarrow$ `if 0 == 0 then 7 else gcd(0, 7 % 0)`

$\rightarrow$ `7`

Another rewriting example:

Consider `factorial`:

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

factorial(4)
```

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)` 3-> $\twoheadrightarrow$ `4 * factorial(3)`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)` 3-> $\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)` 3-> $\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)` 3-> $\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)` 3-> $\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

Another rewriting example:

Consider `factorial`:

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

`factorial(4)`

$\rightarrow$ `if 4 == 0 then 1 else 4 * factorial(4 - 1)` 3-> $\twoheadrightarrow$ `4 * factorial(3)`

$\twoheadrightarrow$ `4 * (3 * factorial(2))`

$\twoheadrightarrow$ `4 * (3 * (2 * factorial(1)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * factorial(0)))`

$\twoheadrightarrow$ `4 * (3 * (2 * (1 * 1)))`

$\twoheadrightarrow$ `24`

What are the differences between the two sequences?

Tail Recursion

Implementation Consideration:

If a function calls itself as its last action, the function's stack frame can be reused. This is called *tail recursion*.

$\Rightarrow$ Tail recursive functions are iterative processes.

In general, if the last action of a function consists of calling a function (which may be the same), one stack frame would be sufficient 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:

```
import scala.annotation.tailrec

@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: Tail recursion

Design a tail recursive version of `factorial`.