Functional Programming Principles in Scala

Martin Odersky

September 12, 2012
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
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.
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?

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

\[(a \times x + b) + (c \times x + d) = (x+c)\times x + (b+d)\]

But it does not define an operator to change a coefficient while keeping the polynomial the same!
For instance the theory of polynomials defines the sum of two polynomials by laws such as:

\[(a \cdot x + b) + (c \cdot x + d) = (x+c) \cdot x + (b+d)\]

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

*Other example:*

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

\[(a \text{ ++ } b) \text{ ++ } c = a \text{ ++ } (b \text{ ++ } c)\]

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

Let’s

- concentrate on defining theories for operators,
- minimize state changes,
- treat operators as functions, often composed of simpler functions.
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.

In particular, functions can be values that are produced, consumed, and composed.

All this becomes easier in a functional language.
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.

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 operators to compose functions
Some functional programming languages

In the restricted sense:

- Pure Lisp, XSLT, XPath, XQuery, FP
- Haskell (without I/O Monad or UnsafePerformIO)

In the wider sense:

- Lisp, Scheme, Racket, Clojure
- SML, Ocaml, F#
- Haskell (full language)
- Scala
- Smalltalk, Ruby (!)
<table>
<thead>
<tr>
<th>Year</th>
<th>Language(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1959</td>
<td>Lisp</td>
</tr>
<tr>
<td>1975-77</td>
<td>ML, FP, Scheme</td>
</tr>
<tr>
<td>1978</td>
<td>Smalltalk</td>
</tr>
<tr>
<td>1986</td>
<td>Standard ML</td>
</tr>
<tr>
<td>1990</td>
<td>Haskell, Erlang</td>
</tr>
<tr>
<td>1999</td>
<td>XSLT</td>
</tr>
<tr>
<td>2000</td>
<td>OCaml</td>
</tr>
<tr>
<td>2003</td>
<td>Scala, XQuery</td>
</tr>
<tr>
<td>2005</td>
<td>F#</td>
</tr>
<tr>
<td>2007</td>
<td>Clojure</td>
</tr>
</tbody>
</table>

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.

A comprehensive step-by-step guide

The standard language introduction and reference.
Recommended Book (3)

Scala for the Impatient

A faster paced introduction to Scala for people with a Java background.

The first part of the book is available for free download
Why Functional Programming?

Functional Programming is becoming increasingly popular because it offers an attractive method for exploiting parallelism for multicore and cloud computing.

To find out more, see the video of my 2011 Oscon Java keynote Working Hard to Keep it Simple (16.30 minutes).

The slides for the video are available separately.
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.
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
Here are some simple interactions with the REPL

```scala
scala> 87 + 145
232
```

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

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

scala> 5 * size
10
```
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
Here is the evaluation of an arithmetic expression:

\[(2 \times \pi) \times \text{radius}\]
Example

Here is the evaluation of an arithmetic expression:

\[(2 \times \pi) \times \text{radius}\]

\[(2 \times 3.14159) \times \text{radius}\]
Example

Here is the evaluation of an arithmetic expression:

\[(2 \times \pi) \times \text{radius}\]

\[(2 \times 3.14159) \times \text{radius}\]

\[6.28318 \times \text{radius}\]
Example

Here is the evaluation of an arithmetic expression:

\[(2 \times \pi) \times \text{radius}\]

\[(2 \times 3.14159) \times \text{radius}\]

\[6.28318 \times \text{radius}\]

\[6.28318 \times 10\]
Example

Here is the evaluation of an arithmetic expression:

\[(2 \times \text{pi}) \times \text{radius}\]

\[(2 \times 3.14159) \times \text{radius}\]

6.28318 \times \text{radius}

6.28318 \times 10

62.8318
Definitions can have parameters. For instance:

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

scala> square(2)
4.0

scala> square(5 + 4)
81.0

scala> square(square(4))
256.0

def sumOfSquares(x: Double, y: Double) = square(x) + square(y)
sumOfSquares: (Double,Double)Double
```
Parameter and Return Types

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

```python
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, e.g:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Int</td>
<td>32-bit integers</td>
</tr>
<tr>
<td>Double</td>
<td>64-bit floating point numbers</td>
</tr>
<tr>
<td>Boolean</td>
<td>boolean values true and false</td>
</tr>
</tbody>
</table>
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

\text{sumOfSquares}(3, 2+2)
Example

\text{sumOfSquares}(3, 2+2)
\text{sumOfSquares}(3, 4)
Example

\[
\text{sumOfSquares}(3, 2+2) \\
\text{sumOfSquares}(3, 4) \\
\text{square}(3) + \text{square}(4)
\]
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

\[
\text{sumOfSquares}(3, 2+2) \\
\text{sumOfSquares}(3, 4) \\
\text{square}(3) + \text{square}(4) \\
3 \times 3 + \text{square}(4) \\
9 + \text{square}(4) \\
9 + 4 \times 4
\]
Example

\[
\text{sumOfSquares}(3, 2+2) \\
\text{sumOfSquares}(3, 4) \\
\text{square}(3) + \text{square}(4) \\
3 \times 3 + \text{square}(4) \\
9 + \text{square}(4) \\
9 + 4 \times 4 \\
9 + 16
\]
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

```python
def loop: Int = loop
```

```
loop → 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:

\[
\text{sumOfSquares}(3, 2+2) \\
\text{square}(3) + \text{square}(2+2)
\]
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:

\[
\text{sumOfSquares}(3, 2+2) \\
\text{square}(3) + \text{square}(2+2) \\
3 \times 3 + \text{square}(2+2) \\
9 + \text{square}(2+2)
\]
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:

\[
\text{sumOfSquares}(3, 2+2)
\]
\[
\text{square}(3) + \text{square}(2+2)
\]
\[
3 \times 3 + \text{square}(2+2)
\]
\[
9 + \text{square}(2+2)
\]
\[
9 + (2+2) \times (2+2)
\]
\[
9 + 4 \times (2+2)
\]
\[
9 + 4 \times 4
\]
\[
25
\]
The first evaluation strategy is known as *call-by-value*, the second 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.
Question: Say you are given the following function definition:

```python
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):

<table>
<thead>
<tr>
<th>CBV</th>
<th>CBN</th>
<th>same</th>
<th>#steps</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(2, 3)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(3+4, 8)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(7, 2*4)</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>test(3+4, 2*4)</td>
</tr>
</tbody>
</table>
Call-by-name vs call-by-value

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

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

![Diagram showing the difference between call-by-name (CBN) and call-by-value (CBV) for the given test function.](attachment:diagram.png)
Evaluation Strategies and Termination

August 31, 2012
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
Question: Find an expression that terminates under CBN but not under CBV.
Non-termination example

Let’s define

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

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

**Under CBN:**

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

**Under CBV:**

```python
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:

```scala
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(loop, 1 + 2)

constOne(loop, 1 + 2)
Conditionals and Value Definitions

August 31, 2012
Conditional Expressions

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

It looks like a if-else in Java, but is used for expressions, not statements.

Example:

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

\(x \geq 0\) 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:

- $e \leq e$, $e \geq e$, $e < e$, $e > e$, $e == e$, $e != e$
Here are reduction rules for Boolean expressions (e is an arbitrary expression):

\[
\begin{align*}
!\text{true} & \implies \text{false} \\
!\text{false} & \implies \text{true} \\
\text{true} \&\& e & \implies e \\
\text{false} \&\& e & \implies \text{false} \\
\text{true} \mathbin{|}\| e & \implies \text{true} \\
\text{false} \mathbin{|}\| e & \implies e
\end{align*}
\]

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-else

\[
\text{if } (b) \ e_1 \text{ then } e_2
\]

\[
\text{if } \text{ (true) } e_1 \text{ else } e_2 \quad \rightarrow \quad e_1
\]

\[
\text{if } \text{ (false) } e_1 \text{ else } e_2 \quad \rightarrow \quad e_2
\]
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` for, which is “by-value”. Example:

\[
\begin{align*}
\text{def } z &= 3 + 4 \\
\text{val } x &= 2 \\
\text{val } y &= \text{square}(x)
\end{align*}
\]

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 \(\text{square}(2)\).
Value Definitions and Termination

The difference between \texttt{val} and \texttt{def} becomes apparent when the right hand side does not terminate. Given

\begin{verbatim}
def loop: Boolean = loop
\end{verbatim}

A definition

\begin{verbatim}
def x = loop
\end{verbatim}

is OK, but a definition

\begin{verbatim}
val x = loop
\end{verbatim}

will lead to an infinite loop.
Exercise

Write functions \texttt{and} and \texttt{or} such that for all argument expressions \(x\) and \(y\):

\[
\begin{align*}
\text{and}(x, y) &= x && y \\
\text{or}(x, y) &= x || y 
\end{align*}
\]

(do not use \texttt{||} and \texttt{&&} in your implementation)

What are good operands to test that the equalities hold?
Example: Square roots with Newton’s method
We will define in this session a function

```scala
/** 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:

<table>
<thead>
<tr>
<th>Estimation</th>
<th>Quotient</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$2 / 1 = 2$</td>
<td>1.5</td>
</tr>
<tr>
<td>1.5</td>
<td>$2 / 1.5 = 1.333$</td>
<td>1.4167</td>
</tr>
<tr>
<td>1.4167</td>
<td>$2 / 1.4167 = 1.4118$</td>
<td>1.4142</td>
</tr>
<tr>
<td>1.4142</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
First, define a function which computes one iteration step

```scala
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.
Second, define a function `improve` to improve an estimate and a test to check for termination:

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

def isGoodEnough(guess: Double, x: Double) =
    abs(guess * guess - x) < 0.001
```
Third, define the sqrt function:

```scala
def sqrt(x: Double) = sqrtIter(1.0, x)
```
Blocks and Lexical Scope

August 31, 2012
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 auxiliary functions inside sqrt.
The sqrt Function, Take 2

```python
def sqrt(x: Double) = {
    def sqrtIter(guess: Double, x: Double): Double =
        if (isGoodEnough(guess, x)) 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 `{ ... }`.

  ```scala
  { 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.
Blocks and Visibility

```scala
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?

```scala
val x = 0
def f(y: Int) = y + 1
val result = {
  val x = f(3)
  x * x
} + 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 $\sqrt{\cdot}$ by eliminating redundant occurrences of the $x$ parameter, which means everywhere the same thing:
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) 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

In Scala, semicolons at the end of lines are in most cases optional. You could write

```scala
val x = 1;
```

but most people would omit the semicolon.

On the other hand, if there are more than one statements on a line, they need to be separated by semicolons:

```scala
val y = x + 1; y * y
```
One issue with Scala’s semicolon convention is how to write expressions that span several lines. For instance

```
someLongExpression
+ someOtherExpression
```

would be interpreted as *two* expressions:

```
someLongExpression;
+ someOtherExpression
```
There are two ways to overcome this problem.

You could write the multi-line expression in parentheses, because semicolons are never inserted inside (...):

```
(someLongExpression
 + someOtherExpression)
```

Or you could write the operator on the first line, because this tells the Scala compiler that the expression is not yet finished:

```
someLongExpression + someOtherExpression
```
You have seen simple elements of functional programming in Scala.

- arithmetic and boolean expressions
- conditional expressions if-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
One simple rule: One evaluates a function application $f(e_1, \ldots, e_n)$

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

This can be formalized as a \textit{rewriting of the program itself}:

\[
\text{def } f(x_1, \ldots, x_n) = B; \ldots f(v_1, \ldots, v_n) \\
\rightarrow \\
\text{def } f(x_1, \ldots, x_n) = B; \ldots [v_1/x_1, \ldots, v_n/x_n] B
\]

Here, \([v_1/x_1, \ldots, v_n/x_n] B\) means:

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

\([v_1/x_1, \ldots, v_n/x_n]\) is called a \textit{substitution}. 

Consider \( \text{gcd} \), the function that computes the greatest common divisor of two numbers.

Here’s an implementation of \( \text{gcd} \) using Euclid’s algorithm.

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

\text{gcd}(14, 21) \text{ is evaluated as follows:}

\text{gcd}(14, 21)
Rewriting example:

gcd(14, 21) is evaluated as follows:

\[
gcd(14, 21) \\
→ \text{if } (21 == 0) \text{ 14 else gcd(21, 14 \% 21)}
\]
Rewriting example:

\[ \text{gcd}(14, 21) \] is evaluated as follows:

\[ \text{gcd}(14, 21) \]

\[ \rightarrow \text{if } (21 == 0) 14 \text{ else gcd}(21, 14 \ % \ 21) \]

\[ \rightarrow \text{if } (\text{false}) 14 \text{ else gcd}(21, 14 \ % \ 21) \]
Rewriting example:

\[ \text{gcd}(14, 21) \text{ is evaluated as follows:} \]

\[ \text{gcd}(14, 21) \]
\[ \rightarrow \text{if } (21 == 0) 14 \text{ else } \text{gcd}(21, 14 \ % \ 21) \]
\[ \rightarrow \text{if } (\text{false}) 14 \text{ else } \text{gcd}(21, 14 \ % \ 21) \]
\[ \rightarrow \text{gcd}(21, 14 \ % \ 21) \]
Rewriting example:

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)
Rewriting example:

gcd(14, 21) is evaluated as follows:

\[
gcd(14, 21)
\rightarrow \text{if (21 == 0) 14 else gcd(21, 14 \% 21)}
\rightarrow \text{if (false) 14 else gcd(21, 14 \% 21)}
\rightarrow \text{gcd(21, 14 \% 21)}
\rightarrow \text{gcd(21, 14)}
\rightarrow \text{if (14 == 0) 21 else gcd(14, 21 \% 14)}
\]
Rewriting example:

\[ \text{gcd}(14, 21) \text{ is evaluated as follows:} \]

\[ \text{gcd}(14, 21) \]
\[ \rightarrow \text{if } (21 == 0) 14 \text{ else } \text{gcd}(21, 14 \% 21) \]
\[ \rightarrow \text{if } (\text{false}) 14 \text{ else } \text{gcd}(21, 14 \% 21) \]
\[ \rightarrow \text{gcd}(21, 14 \% 21) \]
\[ \rightarrow \text{gcd}(21, 14) \]
\[ \rightarrow \text{if } (14 == 0) 21 \text{ else } \text{gcd}(14, 21 \% 14) \]
\[ \rightarrow \text{gcd}(14, 7) \]
Rewriting example:

\[ \gcd(14, 21) \] is evaluated as follows:

\[ \gcd(14, 21) \]
\[ \to \text{if } (21 == 0) \ 14 \ \text{else } \gcd(21, 14 \ % \ 21) \]
\[ \to \text{if } (\text{false}) \ 14 \ \text{else } \gcd(21, 14 \ % \ 21) \]
\[ \to \gcd(21, 14 \ % \ 21) \]
\[ \to \gcd(21, 14) \]
\[ \to \gcd(21, 14) \]
\[ \to \text{if } (14 == 0) \ 21 \ \text{else } \gcd(14, 21 \ % \ 14) \]
\[ \to \gcd(14, 7) \]
\[ \to \gcd(7, 0) \]
Rewriting example:

gcd(14, 21) is evaluated as follows:

\[
gcd(14, 21) \\
\rightarrow \text{if } (21 == 0) 14 \text{ else } gcd(21, 14 \% 21) \\
\rightarrow \text{if } (false) 14 \text{ else } gcd(21, 14 \% 21) \\
\rightarrow gcd(21, 14 \% 21) \\
\rightarrow gcd(21, 14) \\
\rightarrow \text{if } (14 == 0) 21 \text{ else } gcd(14, 21 \% 14) \\
\rightarrow gcd(14, 7) \\
\rightarrow gcd(7, 0) \\
\rightarrow \text{if } (0 == 0) 7 \text{ else } gcd(0, 7 \% 0)
\]
Rewriting example:

gcd(14, 21) is evaluated as follows:

\[
gcd(14, 21) \\
\rightarrow \text{if } (21 == 0) \ 14 \ \text{else} \ \gcd(21, 14 \ % \ 21) \\
\rightarrow \text{if } (\text{false}) \ 14 \ \text{else} \ \gcd(21, 14 \ % \ 21) \\
\rightarrow \gcd(21, 14 \ % \ 21) \\
\rightarrow \gcd(21, 14) \\
\rightarrow \text{if } (14 == 0) \ 21 \ \text{else} \ \gcd(14, 21 \ % \ 14) \\
\rightarrow \gcd(14, 7) \\
\rightarrow \gcd(7, 0) \\
\rightarrow \text{if } (0 == 0) \ 7 \ \text{else} \ \gcd(0, 7 \ % \ 0) \\
\rightarrow 7
\]
Another rewriting example:

Consider factorial:

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

factorial(4)
Another rewriting example:

Consider factorial:

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

```python
factorial(4)
```

→ if (4 == 0) 1 else 4 * factorial(4 - 1)
Consider factorial:

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

`factorial(4)`

→ if (4 == 0) 1 else 4 * factorial(4 - 1)

→ 4 * factorial(3)
Another rewriting example:

Consider factorial:

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

`factorial(4)`

→ if (4 == 0) 1 else 4 * factorial(4 - 1)

→ 4 * factorial(3)

→ 4 * (3 * factorial(2))
Another rewriting example:

Consider factorial:

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

factorial(4)
→ if (4 == 0) 1 else 4 * factorial(4 - 1)
→ 4 * factorial(3)
→ 4 * (3 * factorial(2))
→ 4 * (3 * (2 * factorial(1)))
```
Another rewriting example:

Consider factorial:

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

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))))
Another rewriting example:

Consider factorial:

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

`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)))
Another rewriting example:

Consider factorial:

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

`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)))
→ 120

What are the differences between the two sequences?
Implementation Consideration: If a function calls itself as its last action, the function’s stack frame can be reused. This is called *tail recursion*.

⇒ 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:

```scala
@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.