Functional programming in Scala

Functions

Said BOUDJELDA

Senior Software Engineer @SCIAM
Email : mohamed-said.boudjelda@intervenants.efrei.net
Follow me on GitHub @bmscomp

Course, May 2025

Definition

In functional programming languages, functions are the fundamental building blocks of computation. They are treated differently than in imperative languages, more like mathematical functions and first-class citizens.
We will learn here what that means in practice.

Impure functions

```scala
/**
 * Not pure because it have side affect
 */
val greeting: Unit = println("Hello, Scala")
```

Pure functions

```scala

/**
 * Deterministic: The same input always gives the same
     output.
 * Side effect-free: It does not modify any state or
    interact with the outside world (no I/O, * no
    global variable changes).
 */
val add = (x: Int, y: Int) => x + y

// Same Function with another declaration
val add: (Int, Int) => Int = (a, b) => a + b
```

Functions are not Methods

Methods look and behave very similar to functions, but there are a few key differences between them.

```scala
/**
 * Methods are defined with the def keyword. def is
     followed by a name, parameter list(s), a return
     type, and a body
 */
def add(x: Int, y: Int): Int = x + y
println(add(1, 2)) // 3
```

First-Class Citizens

```scala
/**
 * Be assigned to variables
 * Be passed as arguments
 * Be returned from other functions
 */

val sum = (x:Int , y: Int) => x + y
val double = (x: Int) => x * 2
def application(f: Int => Int, x: Int): Int = f(x)
def apply(f: (Int, Int) => Int, x: Int, y: Int): Int =
    f(x, y)
```

Function Composition

```scala
val f = (x: Int) => x + 1
val g = (x: Int) => x * 2
val h = f.compose(g) // h(x) = f(g(x)) = (x * 2) + 1
```

A lambda (aka anonymous function) is a function defined without a name

```
(x: Int) => x * 2
```

Immutability and Referential Transparency

```scala
// Referentially transparent
def square(n: Int): Int = n * n
// Can be replaced with 16 + 16 or 32
val result = square(4) + square(4)
// Not referentially transparent (depends on external
   state)
```

Recursion and tail call optimization

Recursion is a function that it calls itself

```scala
val factorial: Int => Int = (n: Int) =>
  if (n <= 1) 1            // Base case
  else n * factorial(n - 1) // Recursive case

// Tail call optimization
import scala.annotation.tailrec
val factorial: Int => Int = {
  @tailrec
  def loop(n: Int, acc: Int): Int = n match {
    case 0 | 1 => acc
    case _ => loop(n - 1, n * acc)
  }
  (n: Int) => loop(n, 1)
}
```

Higher-Order Functions (HOFs)

Functions that take other functions as arguments or return them.

- **Abstraction**: Reduce code duplication
- **Expressiveness**: Write more declarative code
- **Composability**: Combine small functions into complex operations
- **Flexibility** : Parameterize behavior

Higher-Order Functions (HOFs)

map function

The map function is a fundamental higher-order function in Scala that applies a given function to each element of a collection and returns a new collection with the transformed elements.

```
/**
 * The map function functiona implementation
 */

val map: [A, B] => (List[A], A => B) => List[B] =
  [A, B] => (list, f) => list match
    case Nil => Nil
    case head :: tail => f(head) :: map(tail, f)

val numbers = List(1, 2, 3, 4)
val doubled = map(numbers, x => x * 2)
//The result will be List(2, 4, 6, 8)
```

flatMap combines mapping and flattening operations. It's more powerful than map because it can handle nested structures and transform each element into a new collection, then flatten the results into a single collection

```
val flatMap:[A, B] => (List[A], A => List[B]) => List[
    B] =
  [A, B] => (list, f) => list.foldRight(List.empty[B])
      ((a, acc) => f(a) ++ acc)

val numbers = List(1, 2, 3)
val result = flatMap(numbers)(n => List(n, n * 10))
// result: List[Int] = List(1, 10, 2, 20, 3, 30)
```

Higher-Order Functions (HOFs)

map vs flatMap

Feature	`map`	`flatMap`
Return type	Single element	Collection/Monadic type
Result structure	Wrapped in same context	Flattened structure
Function signature	`A => B`	`A => F[B]`
Use case	Simple transformations	Transform + flatten
Nested collections	Preserves nesting	Removes one level of nest
For-comprehensions	Used with `yield`	Used with <-

- Both preserve the original collection
- Both are higher-order functions
- `flatMap` is monadic bind operation

Higher-Order Functions (HOFs)

HOFs Filter function

```scala

/**
 * The filter function
 */

val filter: List[Int] => (Int => Boolean) => List[Int] =
  list => predicate => list match
    case Nil => Nil
    case head :: tail =>
      if (predicate(head)) head :: filter(tail)(predicate)
      else filter(tail)(predicate)
```

A **fold** function (also known as **reduce** or **aggregate**) is a powerful higher-order function in functional programming that reduces a collection (like a **List**) to a single value by applying a binary operation repeatedly, starting from an initial value.

HOFs/Fold Function

foldLeft implementation

foldLeft processes elements left-to-right (from the first to the last).

```scala

// foldLeft (Tail-recursive, stack-safe)
def left[A, B](list: List[A])(ini: B)(op: (B, A) => B): B =
  list match
    case Nil => initial
    case head :: tail => left(tail)(op(initial, head))(op)

val nums = List(1, 2, 3, 4)
// foldLeft: (((0 - 1) - 2) - 3) - 4 = -10
nums.foldLeft(0)(_ - _)
```

HOFs Fold Function

foldRight implementation

foldRight processes elements right-to-left (from the last to the first).

```scala
// foldRight (Not tail-recursive)
def foldRight[A, B](list: List[A])(initial: B)(op: (A,
    B) => B): B =
  list match
    case Nil => initial
    case head :: tail => op(head, foldRight(tail)(
        initial)(op))

val nums = List(1, 2, 3, 4)
// foldRight: 1 - (2 - (3 - (4 - 0))) = -2
nums.foldRight(0)(_ - _)
```

Folds Performance & Stack Safety Comparison

Both foldLeft and foldRight are higher-order functions that combine elements of a collection into a single result, but they differ in key ways

Feature	`foldLeft`	`foldRight`
Tail-recursive	$\checkmark$	$\times$
Stack-safe on large collections	$\checkmark$	$\times$
Time Complexity	$O(n)$ (iterative)	$O(n)$ (but stack risk)
Space Complexity	$O(1)$	$O(n)$ (call stack)
Works with infinite streams	$\times$	$\checkmark$ (if lazy)

- $\checkmark =$ Supported, $\times =$ Not supported
- `foldLeft` uses constant space (tail-recursion optimized)
- `foldRight` may overflow on large lists (non-tail-recursive)

Combining Immutability and Referential Transparency

```scala
def double(x: Int): Int = x * 2
def square(x: Int): Int = x * x
val numbers = List(1, 2, 3, 4, 5)
val processed = numbers.map(double).map(square)

// We can also compose
val composed = numbers.map(x => square(double(x)))
```

Partial applied functions

A partial function is a function that is valid for only a subset of values of those types you might pass in to it

Do not confuse partial functions with partially applied functions

```
val root: PartialFunction[Double,Double] =
  case d if (d >= 0) => math.sqrt(d)

root(-1) // will result scala.MatchError
root.isDefinedAt(-1) // Will result false
root(3) // Double = 1.7320508075688772

// List of only roots which are defined
List(0.5, -0.2, 4).collect(root)
```

Currying functions

Currying is a functional programming technique that transforms a function taking multiple arguments into a sequence of functions that each take a single argument.

```scala
// Regular multi-argument function
val add: (Int, Int) => Int = (a, b) => a + b

// Curried version
val addCurried: Int => Int => Int = a => b => a + b

// Usage
val add5: Int => Int = addCurried(5)
println(add5(10))  // 15
```

Closures and Lexical Scope

A closure is a function that "closes over" (captures) variables from its surrounding lexical scope, even if those variables are no longer in scope when the function is executed.

```scala

val outer: Int => () => Int = x => {
  val captured = x
  () => captured + 10
}

val closure = outer(5)
println(closure())  // Output: 15
```

Pure Functions: Key Papers

[Church, 1936] Church, A. *An Unsolvable Problem of Elementary Number Theory*. 1936.
Lambda calculus as basis for pure functions

[Backus, 1978] Backus, J. *Can Programming Be Liberated from the von Neumann Style?* 1978.
FP manifesto emphasizing purity

[Hughes, 1989] Hughes, J. *Why Functional Programming Matters*. 1989.
Purity enables referential transparency

[Odersky, 2014] Odersky, M. *Scala: Unified OOP-FP*. 2014.
Implementing purity in impure environments

[Peyton Jones, 2003] Peyton Jones, S. *Haskell 98 Language Report*. 2003.
Pure-by-default design

[Cherny, 2020] Cherny, E. *Programming TypeScript*. 2020.
Practical purity in TypeScript/JavaScript

Key Papers on Higher-Order Functions in Scala I

[1] Odersky, M., et al. *An Overview of the Scala Programming Language*, 2004. (Introduces first-class HOFs in Scala's OOP/FP blend)

[2] Oliveira, B. C. d. S., et al. *Type Classes as Objects and Implicits*, OOPSLA 2010. (HOFs + implicits for Haskell-style type classes)

[3] Kossakowski, G., et al. *Miniboxing: Improving Specialization*, SCALA 2012. (Optimizing generic HOFs via `@specialized`)

[4] Burmako, E. *Scala Macros: Let Our Powers Combine*, SCALA 2013. (Macros for HOF inlining, e.g., `map`→`while`)

[5] Brachthäuser, J. I. *Effekt: Capability-Passing*, POPL 2020. (HOFs for effect systems beyond monads)

[6] Twitter Engineering. *Functional Programming at Twitter*, 2011. (Scala's `map`/`flatMap` in distributed systems)