# Using Lambda, Map, Filter, and Fold in Scala

Lesson By: Yiran Yu 1008838180

# Table of Contents



## Introduction

### Why Scala?

- Based on the Java Virtual Machine (JVM) and fully compatible with Java
- Combines object-oriented and functional programming paradigms, unlike other purely functional programming languages
- Widely used in **big data** (e.g., Apache Spark), **web development** (e.g., Play Framework), and **distributed systems** (e.g., Akka), even in Netflix and Spotify.

### Scala and Functional Programming

Functional programming in Scala provides powerful tools for concise and expressive data transformation. Among these, **lambda expressions, map, filter, and fold** are essential constructs that enhance code clarity and efficiency, and align with functional paradigms learned throughout CSCC24.

In this lesson, we will explore how to use lambda expressions and higher-order functions like `map`, `filter`, and `fold` in Scala. By the end of this lesson, you will:

- Understand how to define and use lambda expressions in Scala.
- Learn how to apply `map`, `filter`, and `fold` to collections.
- See practical examples of these concepts in action.

## Lambda Expressions in Scala

Lambda expressions, or **anonymous functions**, are functions defined without a name, allows passing functions as arguments without explicit definitions.

**Syntax:**

```
val <lambda_exp> = (<variable>: Type) => <Transformation_Expression>
```

**Example:**

```
// Similar to Haskell, Scala allows type inference:
// val double: Int => Int = _ * 2
val double = (x: Int) => x * 2
println(double(5)) // Output: 10
```

## Higher-Order Functions: Map, Filter, and Fold

### 1. `map` – Transforming Collections

`map` takes some lambda expression as the only parameter, applies the function to every element of the source collection, and returns a new collection of the **same type** as the source collection.

**Syntax:**

```
collection = (e1, e2, e3, ...)

// func is some function
collection.map(func)

// returns collection(func(e1), func(e2), func(e3), ...)
```

**Example 1: Squaring Lists Using `map`**

```
val numbers = List(1, 2, 3, 4)
val squared = numbers.map(x => x * x)
println(squared) // Output: List(1, 4, 9, 16)
```

**Example 2: Processing User Data**

```
case class User(name: String, age: Int)

val users = List(User("Alice", 25), User("Bob", 30), User("Charlie", 35))
val names = users.map(user => user.name)
println(names) // Output: List(Alice, Bob, Charlie)
```

### 2. `filter` – Selecting Elements

`filter` takes in a predicate that can be applied to each element of the collection, and returns a new collection of all the elements that satisfies the given predicate.

**Syntax:**

```
// p accepts any type A and returns a Boolean value
def filter(p: (A) => Boolean): List[A]
```

**Example 1: Filter even numbers in a List**

```
val numbers = (1 to 10).toList // Creating a list from 1 to 10
// Note the use of "_" here, similar to Haskell, it matches everything and
binds nothing
val evens = numbers.filter(_ % 2 == 0)
println(evens) // Output: List(2, 4, 6, 8, 10)
```

**Using `view` can optimize filtering by applying transformations lazily:**

```
val lazyEvens = numbers.view.filter(_ % 2 == 0).toList
println(lazyEvens) // Output: List(2, 4, 6, 8, 10)
```

**Example 2: Filtering Active Users**

```
case class User(name: String, isActive: Boolean)

val users = List(User("Alice", true), User("Bob", false), User("Charlie",
true))
val activeUsers = users.filter(_.isActive)
println(activeUsers) // Output: List(User(Alice, true), User(Charlie,
true))
```

## 3. `fold` – Reducing a Collection to a Single Value

`fold` (`foldLeft` and `foldRight`) takes an initial value and a binary function, iterates through the list from left to right and right to left respectively, and applies the function to the accumulated result and each element. (Equivalent to `foldl` and `foldr` in Racket and Haskell)

**Syntax:**

```
// We use capital letters A and B to indicate 'any' type, while in
Haskell, we use lowercase letters (e.g. a, b)
def foldLeft[B](z: B)(op: (B, A) => B): B
def foldRight[B](z: B)(op: (A, B) => B): B
```

**Example 1: Multiplying a List of Numbers**

```
val numbers = List(1, 2, 3, 4)
val product = numbers.foldRight(1)(_ * _)
println(product) // Output: 24
```

**Example 2: Calculating Total Salary**

```
case class Employee(name: String, salary: Double)

val employees = List(Employee("Alice", 50000), Employee("Bob", 60000),
Employee("Charlie", 70000))
val totalSalary = employees.foldLeft(0.0)((acc, emp) => acc + emp.salary)
println(totalSalary) // Output: 180000.0
```

## Combining Functional Constructs by Chaining Higher-Order Functions

```
val numbers = List(1, 2, 3, 4, 5, 6)
val result = numbers
  .filter((x: Int) => x % 2 == 0)    // Filter even numbers
  .map((x: Int) => x * x)            // Square each number
  .foldLeft(0)((acc, x) => acc + x)  // Sum the results
println(result) // Output: 56
```

## Conclusion and Comparisons

In this lesson, we explored how to use **lambda expressions**, **map**, **filter**, and **fold** in Scala. Here is a summary table of the **differences** between Racket, Haskell and Scala.

| Feature | Racket | Haskell | Scala |
|---|---|---|---|
| **Lambda Expressions** | Syntax: `(lambda (x) (+ x 1))` | Syntax: `\x -> x + 1` | Syntax: `(x: Int) => x + 1` |
| **Map** | Syntax: `(map f lst)` | Syntax: `map f lst` | Syntax: `list.map(f)` |
| **Filter** | Syntax: `(filter pred lst)` | Syntax: `filter pred lst` | Syntax: `list.filter(pred)` |
| **Fold** | Syntax: `(foldl/foldr f init lst)` | Syntax: `foldl/foldr f init lst` | Syntax: `list.foldLeft/foldRight(init)(f)` |
| **Type System** | Dynamically typed. | Statically typed with type inference. | Statically typed with type inference. |
| **Lazy Evaluation** | - Strict evaluation by default. | - Lazy evaluation by default. | - Strict evaluation by default, supports lazy evaluation. |

**Similarities among Racket, Haskell and Scala:**

- Functions are first-class citizens.
- All data structures are immutable by default.

# References

1. *Anonymous functions*. (n.d.). Scala Documentation. https://docs.scala-lang.org/scala3/book/fun-anonymous-functions.html#inner-main
2. GeeksforGeeks. (2019, March 14). *Scala | map() method*. GeeksforGeeks. https://www.geeksforgeeks.org/scala-map-method/
3. *Scala Collections - Filter Method*. (n.d.). https://www.tutorialspoint.com/scala_collections/scala_collections_filter.html
4. Egima, B. (2024, March 18). *Folding lists in scala*. Baeldung on Scala. https://www.baeldung.com/scala/folding-lists