A Tour on Functional Programming in Scala

Nancy Hitschfeld
Matías Toro

OOP is not the silver bullet!

Great mechanisms, but not suitable in all cases

Hard to use effectively

Know how to combine styles appropriately

Functional Programming

Functions are fundamental in creating programs.

Some concepts:

- Immutability
- Pure functions
- Composition
- Anonymous functions
- Higher-order functions
- Clausures
- Currying
- Map
- Fold
- Filter
- Collect
- Zip
- FlatMap
- Pattern matching

Immutability

In functional programming we want every value to remain unchanged.

This leads to thread safe programming.

If we want to modify an object, we create a new one.

```
object Practice{
  case class Asteroid(name: String, diameter: Double)

  val a: Asteroid = Asteroid("1 Ceres", 939.4)
  val aChanged: Asteroid = a.copy(diameter = 941.2)

}
```

Pure function

They should return the same value of the same inputs, and lack of side effects.

> **Referential transparency**: We can replace the piece of code with the resulting value and vice-versa without changing the meaning or the result of our program.

```
def add(a:Int, b:Int) = a + b

val nine = add(4,5)
val eighteen = nine + nine
val eighteen_2 = add(4,5) + add(4,5)
val eighteen_3 = 9 + add(4,5)
```

First-class functions

Functions are first-class data values, which implies their ability to be stored in variables, utilized as arguments in functions, and dynamically generated akin to other values.

Anonymous functions

Functions that has no name.

```
val addOne = (x: Int) => x + 1
addOne(1) // 2
```

```
val add = (x: Int, y: Int) => x + y
add(1, 2) // 3
```

```
val getTheAnswer = () => 42
getTheAnswer() // 42
```

Higher-order functions

Motivation

```
def sumInts(a: Int, b: Int): Int =
  if (a > b) 0 else a + sumInts(a + 1, b)

def cube(x: Int): Int = x * x * x

def sumCubes(a: Int, b: Int): Int =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)

def sumFactorials(a: Int, b: Int): Int =
  if (a > b) 0 else factorial(a) + sumFactorials(a + 1, b)
```

Higher-order functions

HOF: Functions that take/return other functions as parameters.

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if (a > b) 0
  else f(a) + sum(f, a + 1, b)
```

```
def id(x: Int): Int = x

def sumInts(a: Int, b: Int) = sum(id, a, b)

def sumCubes(a: Int, b: Int) = sum(cube, a, b)

def sumFactorials(a: Int, b: Int) = sum(factorial, a, b)
```

Closures

Functions that depends on variables in scope

```
val kelvin = 273.15

val calcTemp = (temp: Double) => temp + kelvin

def main(args: Array[String]): Unit = {
  calcTemp(32)
}
```

Currying

Re-arrange a method into a chain of calls.

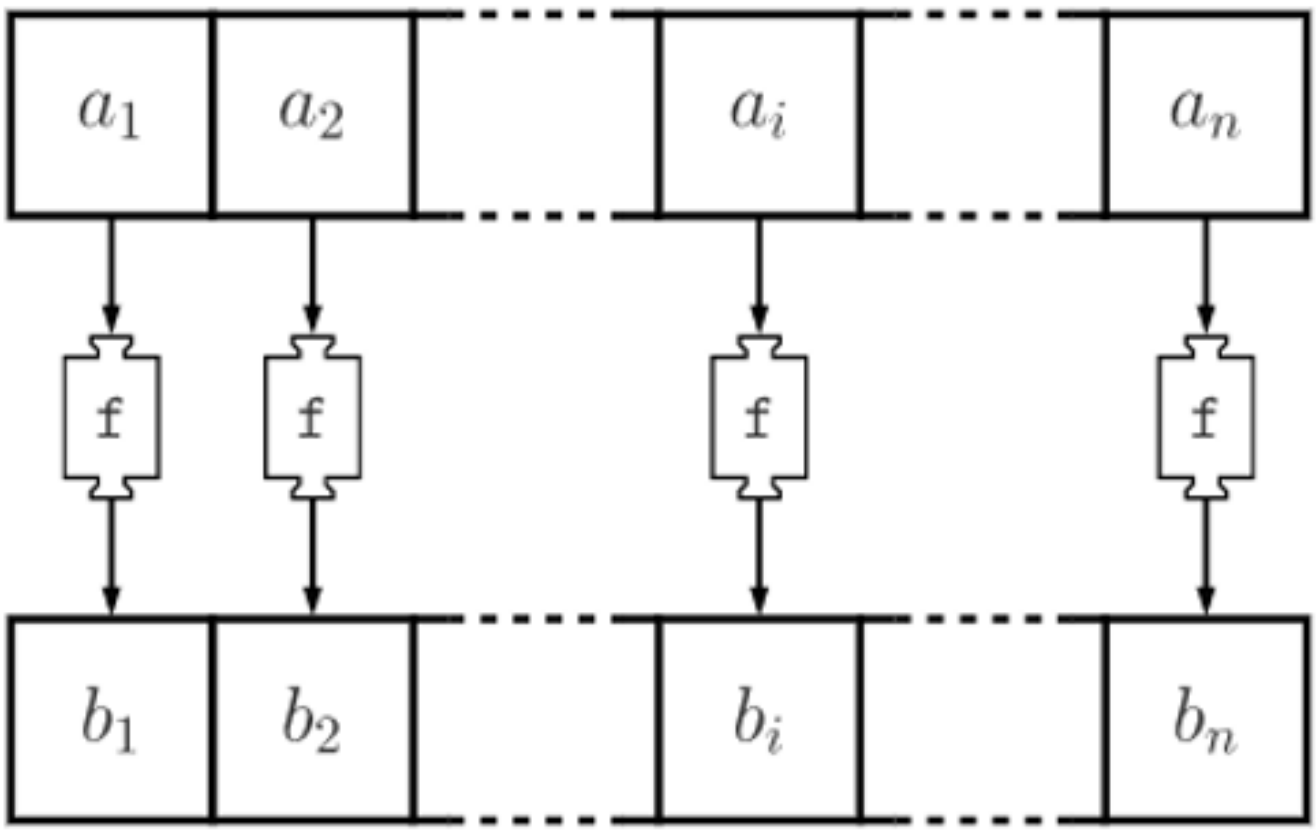
```
def sum(x: Int, y: Int) = x + y
sum(1,2) // 3

def curriedSum(x: Int)(y: Int) = x + y
curriedSum(1)(2) // 3
```

```
val increment: Int => Int = curriedSum(1)
increment(41) // 42
```

Immutable collections

Immutable collections: map

Creates a new collection by applying a function to each element of the initial collection.



Immutable collections: map

Creates a new collection by applying a function to each element of the initial collection.

```
trait Collection[A]{
  def map[B](f: A => B): Collection[B]
}
```

```
List(1,2,3).map(x => x + 1) // List(2,3,4)
Seq(1,2,3).map(x => x * 2) // List(2,4,6)
List(1,2,3).map(Math.sqrt(_))
```

Immutable collections: map

```
trait Collection[A]{
  def map[B](f: A => B): Collection[B]
}
```

```
Seq(0,1,2,3).map{ n => Seq(n, 2*n)}
```

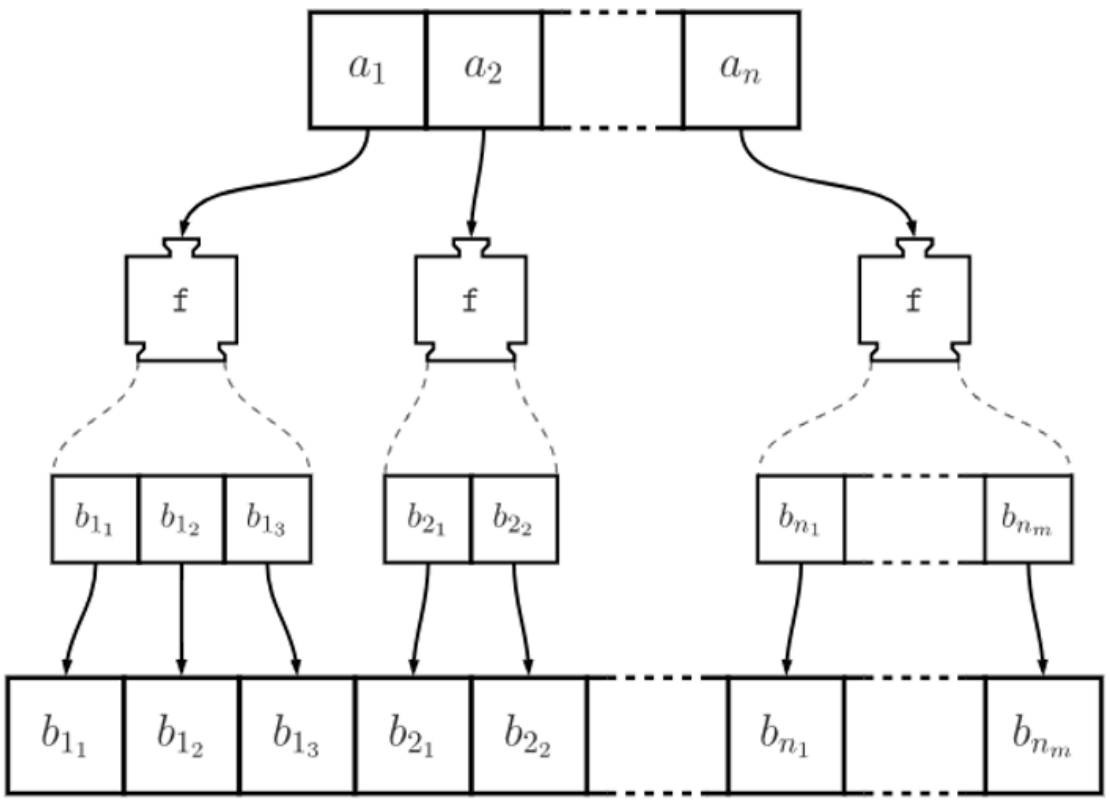
List(List(0, 0), List(1, 2), List(2, 4), List(3, 6))

```
Seq(0,1,2,3).map{ n => Seq(n, 2*n)}.flatten
```

List(0, 0, 1, 2, 2, 4, 3, 6)

Immutable collections: flatMap

Creates a new collection by applying a function “f” to every element and all sub elements formed are “flattened” into one single resulting collection.



Immutable collections: flatMap

Creates a new collection by applying a function “f” to every element and all sub elements formed are “flattened” into one single resulting collection.

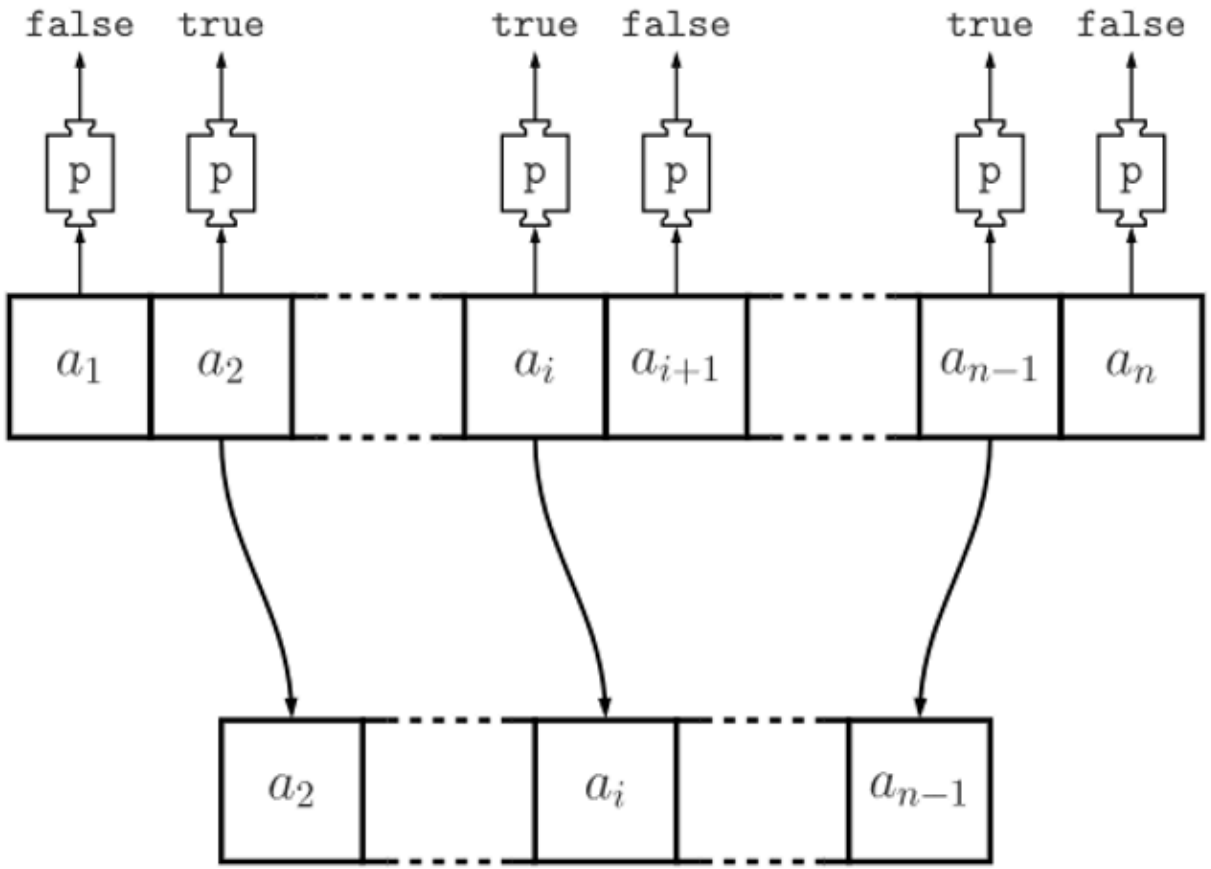
```
trait FlatMapCollection[A]{
  def flatMap[B](f: A => FlatMapCollection[B]): FlatMapCollection[B]
}
```

```
Seq(0,1,2,3).flatMap{ n => Seq(n, 2*n)}
```

List(0, 0, 1, 2, 2, 4, 3, 6)

Immutable collections: filter

Constructs a collection having only the elements which satisfy a condition or predicate, while old values not corresponding are deleted.



Immutable collections: filter

Constructs a collection having only the elements which satisfy a condition or predicate, while old values not corresponding are deleted.

```
trait Filter[A]{
  def filter(predicate: A => Boolean): Filter[A]
}
```

```
Seq(0,1,2,3).filter{ n => n%2==0 }
```

List(0, 2)

Immutable collections: filter

```
List("x", "y", "2", "3", “a")
    .filter(s => "abcdefghijklmnopqrstuvwxyz".contains(s))
```

List(x, y, a)

```
val isNonEmpty = (s:String) => s.size > 0
List("abc", "", "d").filter(isNonEmpty)
```

List(abc, d)

Exercise #1

Transform a list of sentences into a list of words.

Hint: `split(x: String):List[String]` is a method of String objects that breaks the string by delimiter `x`.

```
val sentences = List("Hello world", "", "I love
functional programming")

def getWords(sentences: List[String]): List[String] = {
    ???
}

println(getWords(sentences))
```

List(Hello, world, I, love, functional, programming)

Exercise #1

Transform a list of sentences into a list of words.

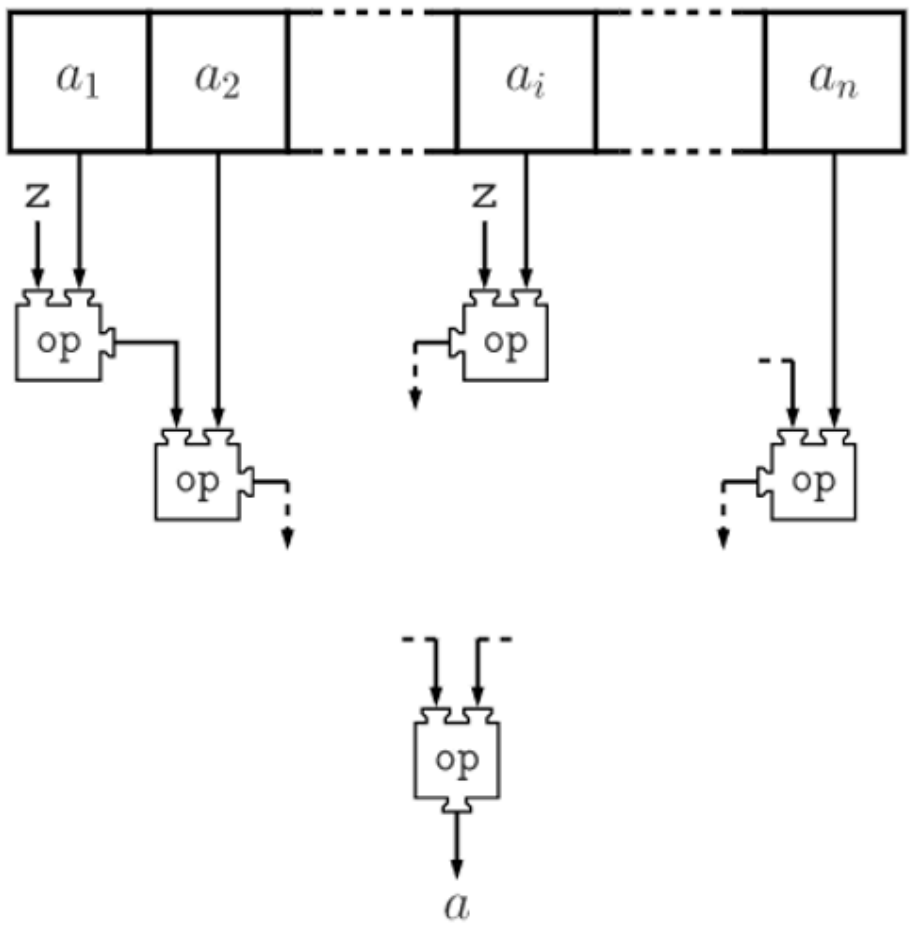
Hint: `split(x: String):List[String]` is a method of String objects that breaks the string by delimiter `x`.

```
val sentences = List("Hello world", "", "I love
functional programming")

def getWords(sentences: List[String]): List[String] = {
  sentences.flatMap(_.split(" ")).filter(isNonEmpty)
}

println(getWords(sentences))
```

List(Hello, world, I, love, functional, programming)

Immutable collections: fold/reduce

Given an initial value, it applies a binary operator to pairs of elements (taken as tuple) successively until returns the result needed



Immutable collections: fold/reduce

Given an initial value, it applies a binary operator to pairs of elements (taken as tuple) successively until returns the result needed

```
trait FoldedCollection[A,B]{
  def fold(seed: B)(operator: (B,A) => B): B
}
```

```
val names = Seq("Andrei","Vlad","Ryad")
names.foldLeft(0)((acc,el) => acc+el.length)
```

14

Exercise #2

Use foldLeft to compute the max of a given list of natural numbers ℕ.

```
List(10,4,31,2)
   .foldLeft(0)((acc,x) => Math.max(acc, x))
```

31

Exercise #3

Implement map using foldLeft.

Hint: Use "`list :+ element`" to append an element to the end of a list.

```
def myListMap[A,B](xs: List[A], f: A => B): List[B] = {
  xs.foldLeft(List[B]())((acc, x) => acc :+ f(x))
}
myListMap(List(1,2,3), (x:Int) => x+1)
```

List(2, 3, 4)

This operator is O(n), so the whole function is O(n^2)

Exercise #3.5

Implement map using **foldRight**.

Hint: Use "`element :: list`" to append an element to the beginning of a list.

The acc is on the right now

```
def myListMap[A,B](xs: List[A], f: A => B): List[B] = {
  xs.foldRight(List[B]())((x, acc) => f(x) :: acc)
}
myListMap(List(1,2,3), (x:Int) => x+1)
```

This operator is O(1), so the whole function is O(n)

Exercise #4

Write a function that inverts a list

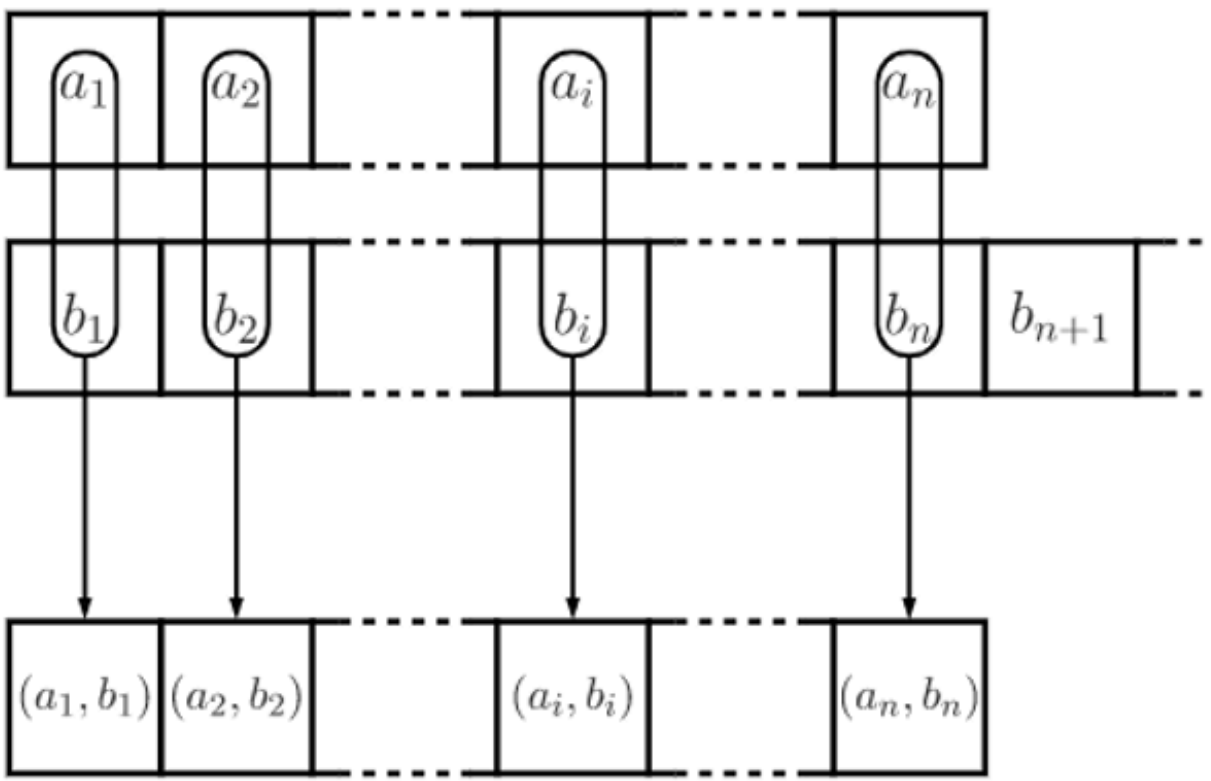
Hint: Use "`element :: list`" to append an element to the beginning of a list.

```
def invert[A](xs: List[A]) : List[A] = {
  xs.foldLeft(List[A]())((acc, x) => x :: acc)
}
invert(List(1,2,3))
```

List(3, 2, 1)

Immutable collections: zip

Creates a whole new collection by pairing each element with another element from a second collection which has the same position/index, discarding unpaired tuples.



Immutable collections: zip

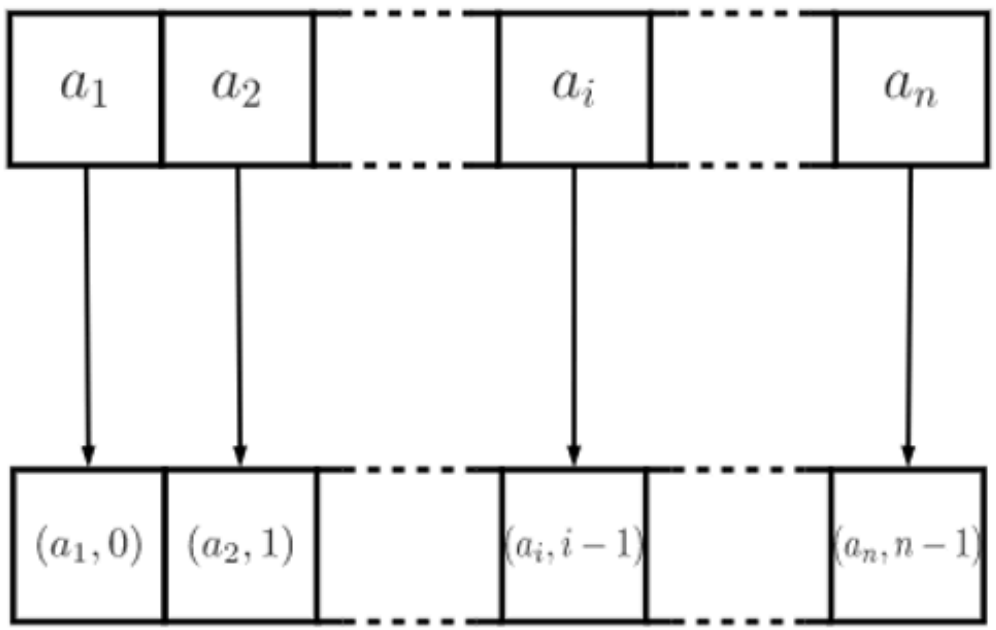
Creates a whole new collection by pairing each element with another element from a second collection which has the same position/index, discarding unpaired tuples.

```
trait ZipCollection[A]{
  def zip[B](bs: ZipCollection[B]): ZipCollection[(A,B)]
}
```

```
val names = Seq("Andrei","Vlad","Ryad")
val scores = List(80,97,23)
names.zip(scores)
```

List((Andrei,80), (Vlad,97), (Ryad,23))

Immutable collections: zipWithIndex

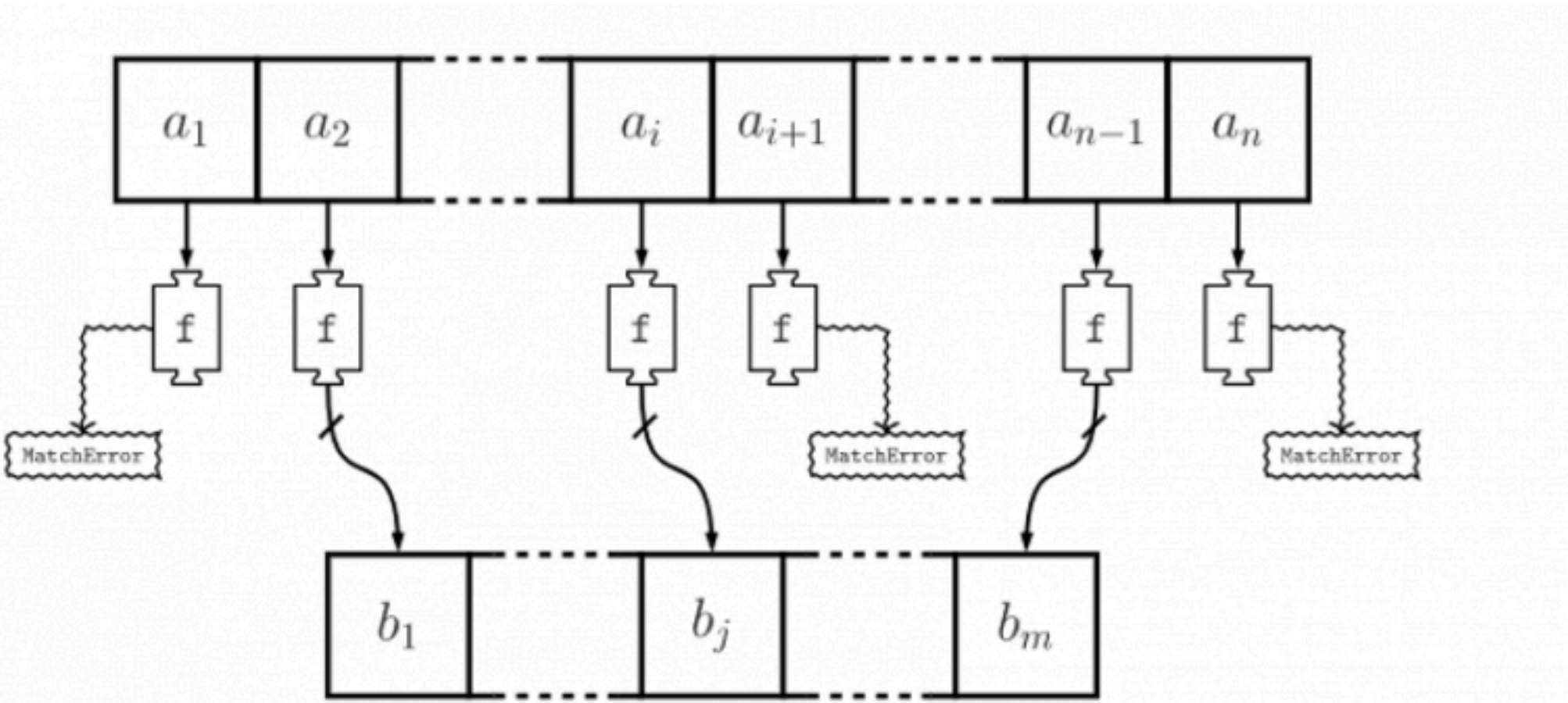



```
val names = Seq("Andrei","Vlad","Ryad")
names.zipWithIndex
```

List((Andrei,0), (Vlad,1), (Ryad,2))

Immutable collections: collect

Build a collection with elements as a result of applying a partial function “f” on them and discarding the rest.



Immutable collections: collect

Build a collection with elements as a result of applying a partial function “f” on them and discarding the rest.

```
trait Collect[A] {
  def collect[B](f: PartialFunction[A,B]): Collect[B]
}
```

```
val divide: PartialFunction[Int,Int] = {
  case d if d!= 0 => 42/d
}
List(0,1,2).collect{divide}
```

List(42, 21)

A lot More to Talk About!

License

Based on tutorial https://4mayo.ro/2022/11/02/the-complete-guide-to-scala-functional-programming-24-code-examples/