```clojure
(defn learner? [person] (= :learner (:type person)))
(defn filter-persons [people] (filter learner? people))
```

```scala
case class Message(sender: String, subject: String, body: String)
val communityMessage = Message("alex+yoan", "Func lang on jvm", "Come as you are")
```

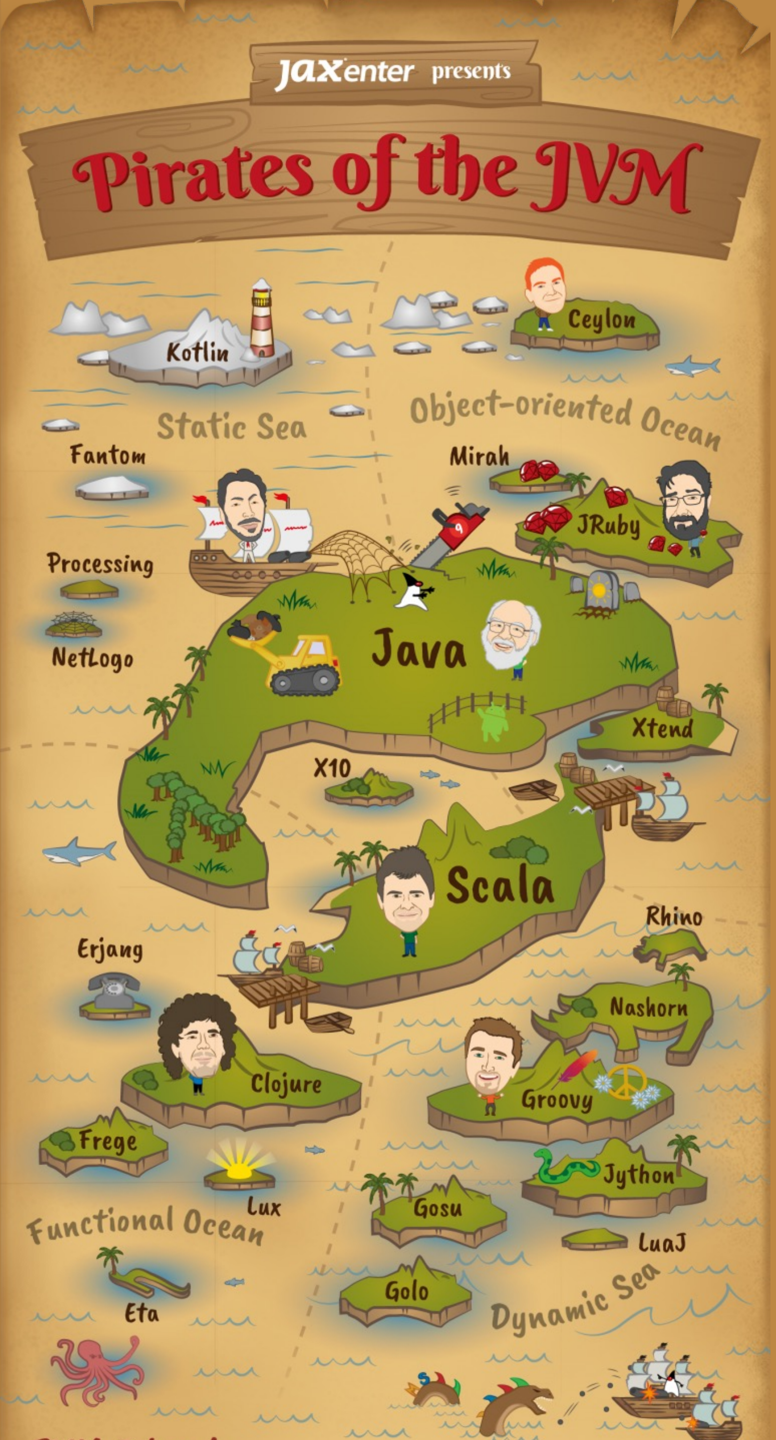
```kotlin
fun whichDate(event: String): Date {
    return when (event) {
        "CoolEvent on FP" -> Date(2021,9,8)
        else -> throw IllegalArgumentException("Not cool enough !!!")
    }
}
```

Functional languages on the jvm 101

08/09/2021 at 2:00 PM

Scala

Kotlin



JVM Islands

Java
Discovered: 1995
Discoverer: James Gosling
Climate: Object-oriented, statically typed, lambda expressions
Population: Code Joes, open source lovers, MS haters
National Hymn: Write once, run everywhere

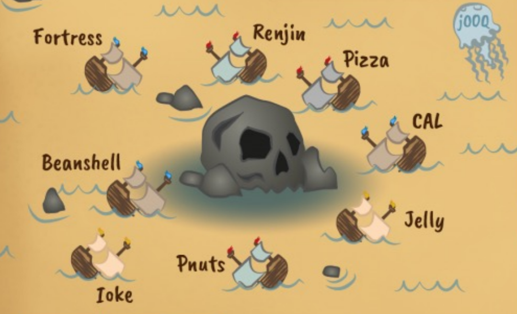
Groovy
Discovered: 2003
Discoverer: James Strachan
Climate: Dynamic-oriented, imperative, scripting
Population: Ironically, very serious company coders
National Hymn: Making Java Groovy

JRuby
Discovered: 2002
Discoverer: Jan Arne Petersen, Charles Nutter, Thomas Enebo
Climate: Dynamic, reflective, object-oriented
Population: Narcissists and beauty queens

Kotlin
Discovered: 2016
Discoverer: Dimitry Jemerow and JetBrains crew
Climate: Object-oriented, static typing
Population: Pioneers, adventurous hipsters
National Hymn: Programming for the JVM, Android and the browser

Scala
Discovered: 2004
Discoverer: Martin Odersky
Climate: Static, multi-paradigm, traits, pattern matching
Population: Programming geniuses and aristocrats
National Hymn: Object-oriented meets functional

Clojure
Discovered: 2007
Discoverer: Rich Hickey
Climate: Functional, dynamic, static typing discipline
Population: Emigrants from Lisp
National Hymn: Treat code as data



Frege
Discovered: 2011
Discoverer: Ingo Wechsung
Climate: Functional, static, strong, inferred
Population: Explorers, adventurers, functional purists
National Hymn: A Haskell for the JVM

Jython
Discovered: 2001
Discoverer: Barry Warsaw
Climate: Dynamic, duck typing
Population: Picky programmers in a rush
National Hymn: Python for the Java Platform

Nashorn
Discovered: 2011
Discoverer: Jim Laskey
Climate: Dynamic, JavaScript engine for the JVM
Population: Bipolar programmers, Java and JavaScript lovers
National Hymn: The Raging Red Rhinoceros

Ceylon
Discovered: 2011
Discoverer: Gavin King, Red Hat
Climate: Object-oriented, static, compiles to JVM and JavaScript
Population: Static typing fetishists
National Hymn: Say more, more clearly

Golo
Discovered: 2012
Discoverer: INSA lyon
Climate: Dynamic, weak typing, foundation for programming and language derivative experiments

Xtend
Discovered: 2011

Ceylon
Discovered: 2011
Discoverer: Gavin King, Red Hat
Climate: Object-oriented, static, compiles to JVM and JavaScript
Population: Static typing fetishists
National Hymn: Say more, more clearly

Golo
Discovered: 2012
Discoverer: INSA lyon
Climate: Dynamic, weak typing, foundation for programming and language derivative experiments
Population: Scientists who love red wine, baguettes and berets
National Hymn: The world didn't need another JVM language. So we built yet another one. A simple one.

Xtend
Discovered: 2011
Discoverer: Sven Efftinge, Sebastian Zarnekow
Climate: Object-oriented, imperative, functional
Population: Programming do-gooders
National Hymn: Java 10, today!

Gosu
Discovered: 2002
Discoverer: Guidewire Software
Climate: Open type system, statically and dynamically compiles to bytecode
Population: Type tinkerer and pragmatists
National Hymn: Hey look! It's a pragmatic language for the JVM ...

Lux
Discovered: 2014
Discoverer: Eduardo Julian
Climate: Functional, statically-typed
Population: Indiana Jones-like programmers
National Hymn: Great complexity without getting buried by it

Fantom
Discovered: 2005
Discoverer: Brian Frank, Andy Frank
Climate: Static, dynamic C-like language, compiles to JavaScript and runs on the .NET Common Language Runtime (CLR)
Population: Fans that became phantoms
National Hymn: Because Pragmatism Wins the Day!

X10
Discovered: 2004
Discoverer: IBM
Climate: Object-oriented, static, strong, safe, constrained
Population: Sweatshop programmers
National Hymn: Performance and Productivity at Scale

Erjang
Discovered: 2009
Discoverer: Kresten Krab Thorup
Climate: A JVM-based Erlang VM, functional, concurrent
Population: Java-addicted Erlang fans
National Hymn: Everything is a process

Mirah
Discovered: 2008
Discoverer: Charles Oliver Nutter
Climate: Object-oriented, static with dynamic features
Population: Rubyists who love static typing
National Hymn: A new way of looking at JVM languages

LuaJ
Discovered: 2009
Discoverer: James Roseborough
Climate: Dynamic scripting language, fast, portable, embeddable
Population: Gamers and other speed freaks
National Hymn: How small can a lua interpreter be if it's written in Java?

NetLogo
Discovered: 1999
Discoverer: Uri Wilensky
Climate: Modeling
Population: Emigrants from Logo
National Hymn: Simulating natural and social phenomena

Processing
Discovered: 2001
Discoverer: Casey Reas, Benjamin Fry
Climate: Object-oriented language for electronic arts, new media art and visual design communities
Population: Artists, designers, students
National Hymn: To teach programming fundamentals within a visual context

Eta
Discovered: 2017
Discoverer: Typelead
Climate: Pure, lazy, strongly typed, functional
Population: Haskell-is-not-enough thinkers
National Hymn: Bring the benefits of Haskell to the JVM

Sources

Language classification

	Functional	Not functional
Static	Scala Kotlin	Java 8
Dynamic	Clojure	Groovy

Syntax

Kotlin

```kotlin
// VARIABLES
var x = 5
x = 6
// Constants
val const = 5

// String interpolation
val language = "Kotlin"
val sheetName = "Let's learn $language basics"

// Explicit type
val explicit: Double = 5.0
```

Scala

```scala
// VARIABLES
var x = 5
x = 6
// Constants
val x = 5

// String interpolation
val language = "Scala"
val sheetName = s"Let's learn $language basics"

// Explicit type
val x: Double = 5
```

```clojure
; Mutable values (atoms)
(def x (atom 5))
(reset! x 6)

; Immutable values
(def x 5)

(def language "Scala")
(def sheet-name (str "Let's learn " language
                     " basics"))

(def x ^Double 5)
```

Functions

Kotlin

```kotlin
// FUNCTIONS
// Define function : need types for every arg
fun square(x: Int) = x * x

// Use default parameter
fun squareWithDefaultParameter(x: Int = 4) = x * x
```

Scala

```scala
// FUNCTIONS
// Define function : need types for every arg
def square(x: Int) = x * x

// Use default parameter
def squareWithDefaultParameter(x: Int = 4) = x * x
```

```clojure
; FUNCTIONS
; Define function
(defn square [x] (* x x))

; Use default parameter
(defn square
   ([] (square 4))
   ([x] (* x x)))
```

Anonymous Functions

Kotlin

```kotlin
// Anonymous functions
(1..5).map { x -> x * x }

// it arg
(1..5).map { it * 2 }
(1..5).reduce { acc, element -> acc + element }
(1..5).map { it * it }
```

Scala

```scala
// Anonymous functions
(x: Double) => x * x
(1 to 5).map(x => x)

// Underscore is positionally matched arg
(1 to 5).map(_ * 2)
(1 to 5).reduceLeft(_ + _)
(1 to 5).map(x => x * x)
```

```clojure
; Anonymous functions
(fn [x] (* x x))

; passing a function
(map square (range 1 5))

; passing an anonymous function
(map (fn [x] (* x x)) (range 1 5))

; passing shortcut function
(map #(* %1 %1) (range 1 5))
```

```scala
// Call by value
def f(x: R) = x
// Call by name -> lazy parameter
def f(x: => R) = x
```

No return

Kotlin

```kotlin
// Block style returns last expression
(1..5).map { x ->
    val y = x * 2
    println(y)
    y
}

// Pipeline style
(1..5)
    .filter { it % 2 == 0 }
    .map { it * 2 }
```

Scala

```scala
// Block style returns last expression
(1 to 5).map { x =>
  val y = x * 2
  println(y)
  y
}

// Pipeline style (or parens too)
(1 to 5) filter {
  _ % 2 == 0
} map {
  _ * 2
}
```

```clojure
; last value is returned (y)
(map (fn [x]
  (let [y (* x 2)]
     (println y)
     y))
  (range 1 5))

; passing shortcut function
(map #(* %1 2)
   (filter #(zero? (mod %1 2)) (range 1 5)))

; easier to read
(->> (range 1 5)
     (filter #(zero? (mod %1 2)))
     (map #(* %1 2)))
```

Composition / Curying

Kotlin

```kotlin
// Function composition
fun compose(
    g: (x: Double) -> Double,
    h: (x: Double) -> Double
): (Double) -> Double = { x -> g(h(x)) }
fun minus1times2(x: Double) = compose({ it * 2
}, { it - 1 })(x)

println(minus1times2(4.0))

// Currying
fun sum(a: Int, b: Int): Int = a + b
val curriedSum: (Int) -> Int = { sum(it, it) }
curriedSum(1)
```

Scala

```scala
// Function composition
def compose(g: R => R, h: R => R) = (x: R) => g(h(x))
def minus1times2 = compose(_ * 2, _ - 1)
println(minus1times2(4))

// Currying
def sum(x: Int, y: Int): Int = x + y
def curriedSum(x: Int)(y: Int): Int = x + y
def curriedSum2: Int => Int => Int = (sum _).curried
curriedSum(1)(2)
curriedSum2(1)(2)
```

```clojure
; Function composition
(defn minus1-times2 [x]
  (comp #(* x 2) dec))
(println (minus1-times2 4))

; Every function can be currified
; using the function “partial”
(defn sum [x y] (+ x y))
(defn sum2 [x] (partial sum 2))
```

Generic Types

Kotlin

```kotlin
fun <T> mapMake(g: (T) -> T, seq: List<T>) = seq.map(g)
mapMake({ x -> x / 2 }, (1..5).toList())

// Varargs
fun sum(vararg args: Int) = args.reduce { acc, element -> acc + element }
sum(1, 2, 3, 4)
```

Scala

```scala
def mapmake[T](g: T => T)(seq: List[T]) = seq.map(g)
mapmake[Int](x => x / 2)((1 to 5).toList)

// Varargs
def sum(args: Int*) = args.reduceLeft(_ + _)
sum(1, 2, 3, 4)
```

```
// Clojure is dynamic
```

Data Structures

Kotlin

```kotlin
// Tuple literal -> Tuple3
Triple(1, 2, 3)
// Tuple destructuring
var (a, b, c) = Triple(1, 2, 3)

// Immutable list
var xs = listOf(1, 2, 3)
var ys = listOf(4, 5, 6)
// Indexing
xs[2]
// Range
(1..5) == 1 until 6
```

Scala

```scala
// Tuple literal -> Tuple3
(1, 2, 3)
// Tuple destructuring
var (x, y, z) = (1, 2, 3)

// Immutable list
var xs = List(1, 2, 3)
var ys = List(4, 5, 6)
// Indexing
xs(2)
// Cons -> short for construct a new List object
// Only at the beginning of a list
1 :: List(2, 3)
// Range
(1 to 5).equals(1 until 6)
```

```clojure
; Lists / Vectors are tuple
[1 2 3]
(let [[x y z] [1 2 3]]
   (println x))

; Everything is immutable
(def xs [1 2 3])
(def ys [1 2 3])

; Indexing
(get xs 2)

; Cons
(conj '(2 3) 1)
(conj  [1 2] 3)
```

Control constructs

Kotlin

```kotlin
fun happy() = println(";-)")
fun sad() = println(":-(")

// Conditional
if (true) happy() else sad()

// While
while (x < 5) {
    println(x)
    x += 1
}

xs.filter { it % 2 == 0 }.map { it * 10 }
```

Scala

```scala
def happy = println(";-)")
def sad = println(":-(")

// Conditional
if (true) happy else sad

// While
while (x < 5) {
  println(x)
  x += 1
}

// For loop
for (x <- xs if x % 2 == 0)
  yield x * 10
// Same as
xs.filter(_ % 2 == 0).map(_ * 10)
```

```clojure
; Lists / Vectors are tuple
(defn happy [] (println ";-)"))
(defn sad [] (println ":-("))

; Conditional
(if true happy sad)
(when true happy)

; while
(def x (atom …))
(while (< @x 5)
  (printn x)
  (swap! x inc))

; For loop
(for [x xs :when (zero? (mod x 2))]
  (* x 10))

(->> xs
     (filter #(zero? (mod %1 2)))
     (map #(* %1 2)))
```

Pattern Matching

Kotlin

```kotlin
val v42 = 42
when (v42) {
    42 -> println("42")
    // Other cases
    else -> println("Not 42")
}
```

Scala

```scala
val v42 = 42
v42 match {
  case 42 => println("42")
  // Other cases
  case _ => println("Not 42")
}
```

```clojure
(require '[clojure.core.match
:refer [match]])

(def v42 42)
(match [v42]
  [42] (println "42")
  // Other cases
  :else (println "Not 42")
)
```

Object Oriented

Kotlin

```kotlin
// x is only available in class body.
class C1(x: Double)

// Automatic public member defined
class C2(val x: Double)

val c1 = C2(9.0)
c1.x

// interfaces
interface APerson {
    val name: String
}
```

Scala

```scala
// x is only available in class body.
class C(x: R)

// Automatic public member defined
class C(val x: R)

val c1 = new C(9)
c1.x

// Traits -> interfaces
trait APerson {
  def name: String
}
```

```
// lol ?
```

Immutable data structure (Record)

Kotlin

```kotlin
// all constructor parameters are public and immutable
data class Student(override val name: String, val year: Int) : APerson
data class Teacher(override val name: String, val specialty: String) : APerson

// Pattern matching on data classes
fun personToString(p: APerson): String = when (p) {
    is Student -> "${p.name} is a student in Year ${p.year}."
    is Teacher -> "${p.name} teaches ${p.specialty}."
    else -> throw IllegalArgumentException("Not supported person type")
}
```

Scala

```scala
// all constructor parameters are public and immutable
case class Student(name: String, year: Int) extends APerson
case class Teacher(name: String, specialty: String) extends APerson

// Pattern matching on case classes
def personToString(p: APerson): String = p match {
  case Student(name, year) => s"$name is a student in Year $year."
  case Teacher(name, whatTheyTeach) => s"$name teaches $whatTheyTeach."
}
```

```clojure
// all constructor parameters are public and immutable
(defrecord Student [name year])
(defrecord Teacher [name specialty])

(defmulti person-to-string class)
(defmethod person-to-string Student [{:keys [name year]}]
    (str name " is a student in Year " year "."))
(defmethod person-to-string Teacher [{:keys [name specialty]}]
    (str name " teaches " specialty "."))
```

Record equality

```kotlin
// Data equality
val bart1 = Student("Bart Simpson", 2021)
val bart2 = Student("Bart Simpson", 2021)
bart1 == bart2
// case class has an automatically-generated copy method
// when you need to perform the process of a) cloning an object and b)
updating one or more of the fields during the cloning
val homer = bart1.copy(name = "Homer Simpson")
```

```scala
// Case equality
val bart1 = Student("Bart Simpson", 2021)
val bart2 = Student("Bart Simpson", 2021)
bart1.equals(bart2)
// case class has an automatically-generated copy method
// when you need to perform the process of a) cloning an object and b)
updating one or more of the fields during the cloning
val homer = bart1.copy(name = "Homer Simpson")
```

```clojure
// Data equality
(def bart1 (->Student "Bart Simpson" 2021])
(def bart2 (->Student "Bart Simpson" 2021])
(= bart1 bart2)
```

Extension methods

Kotlin

```
fun Int.isEven(): Boolean = this % 2 == 0
listOf(1, 4, 7, 8, 12).map { it.isEven() }
```

Scala

```
extension (i: Int)
  def isEven(): Boolean = i % 2 == 0
List(1, 4, 7, 8, 12).map { _.isEven() }
```

```
; Everything is function

; However you can patch Java Protocols
; but it’s too advanced for today
```

Monads

Kotlin

```
// OPTIONS -> no out of the box
// If you want some you can use -> Arrow :
```

https://arrow-kt.io/docs/patterns/monads/

Scala

```scala
// OPTIONS
val option: Option[Int] = Some(42)
None
Option(null) == None

// Others : Try, Either, Future


// Map it
option.map(f(_))
// equivalent to
option match {
  case Some(x) => Some(f(x))
  case None => None
}
```

Clojure

```clojure
; You have only one option
; use Clojure

; However almost all standard methods can deal
; with nil values pragmatically

; Besides Clojure provides nil punning
(if-let [a 1] (+ a 2))
```

Null Safety

Kotlin

```kotlin
// In Kotlin, the type system distinguishes between references that
// can hold null (nullable references) and those that cannot (non-null references)
var str: String = "abc"
// str = null -> DO NOT COMPILE

// To allow null you must explicitly say it
val nullStr: String? = null

// Safe calls
println(nullStr?.length)

// Perform operation if not null with let
val listWithNulls: List<String?> = listOf("Kotlin", null, "Clojure", null, "Scala")
for (item in listWithNulls) {
    item?.let { println(it) }
}
```

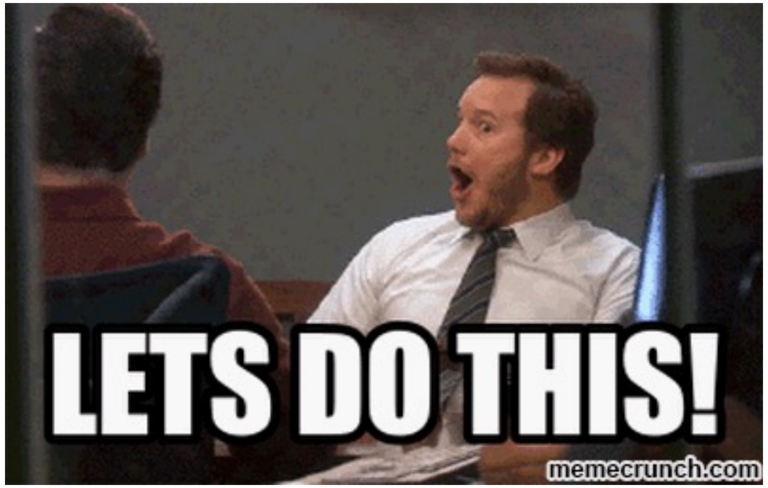
```kotlin
// Instead of this
val length: Int = if (nullStr != null) str.length else -1
// You can use the Elvis operator ?:
val elvisLength = nullStr?.length ?: -1
```

Let's practice

- Clone the repository here : https://github.com/ythirion/fp101.git
- Play a few minutes with the REPL in the syntax script – **5'**

- Open 01.data-structures – **40'**
  - Discover the content of the domain script
  - Implement what is asked in the script by using the syntax you have just seen
  - Follow the implementation instructions if some

- Go further with 02.refactoring



Let's compare solutions

Kotlin

```kotlin
println("2. Who owns Cats ?")
// Create an extension on Person allowing you to filter efficiently
fun Person.hasPetType(type: PetType): Boolean = this.pets.map { it.type }.contains(type)
people.filter { p -> p.hasPetType(PetType.CAT) }
```

Scala

```scala
println("2. Who owns Cats ?")
// Create an extension on Person allowing you to filter efficiently
extension (p: Person)
  def hasPetType(`type`: PetType): Boolean = p.pets.map(_.`type`).contains(`type`)

people.filter(p => p.hasPetType(PetType.CAT))
```

```clojure
(println "2. Who owns Cats ?")
(defn has-pet-type [pet-type person]
(some #{pet-type} (->> person :pets (map :type))))
(filter #(has-pet-type :cat %) people)
```

Let's compare solutions

Kotlin

```kotlin
println("3. What are the names of Mary Smiths cats ?")
people.firstOrNull { "${it.firstName} ${it.lastName}" == "Mary Smith" }
    ?.pets
    ?.filter { it.type == PetType.CAT }
    ?.map { it.name }
```

Scala

```scala
println("3. What are the names of Mary Smiths cats ?")
people.find(p => s"${p.firstName} ${p.lastName}" == "Mary Smith") match {
  case Some(marySmith) => marySmith.pets.filter(_.`type` == PetType.CAT).map(_.name)
  case None => println("Mary smith not found")
}
```

```clojure
(println "3. What are the names of Mary Smiths cats ?")
(defn full-name [person]
(str (:firstname person) " " (:lastname person)))
(->> people (filter #(= "Mary Smith" (full-name %))) first :pets (map :name))
```

Let's compare solutions

Kotlin

```kotlin
println("8. What is the average Pet age ?")
people.flatMap { it.pets }
    .map { it.age }
    .average()
```

Scala

```scala
println("8. What is the average Pet age ?")
// You can define an implicit class to have access to an average function
implicit class ImplSeqInt(values: Seq[Int]) {
  def average = values.map(_.toDouble).sum / values.length
}

people.flatMap(_.pets)
  .map(_.age)
  .average
```

```clojure
(println "8. What is the average Pet age ?")
(defn pets-age [person] (->> person :pets (mapv :age)))
(defn average [coll] (/ (apply + coll) (count coll)))
(->> people (map pets-age) flatten average)
```

Let's compare solutions

Kotlin

```kotlin
println("10. What are the parks in which each person can walk with all their pets ?")
// For each person described as "firstName lastName" returns the list of names possible parks to go for a walk
fun List<Park>.filterFor(person: Person): List<Park> = this.filter { it.authorizedPets.containsAll(person.getPetTypes())
}
people.groupBy { "${it.firstName} ${it.lastName}" }
    .mapValues { t ->
        t.value.map { person ->
            parks.filterFor(person)
                .map { park -> park.name }
        }
    }
```

Scala

```scala
println("10. What are the parks in which each person can walk with all their pets ?")
// For each person described as "firstName lastName" returns the list of names possible parks to go for a walk
extension (parks: Seq[Park])
  def filterFor(person: Person): Seq[Park] = {
    parks.filter(park => person
      .getPetTypes()
      .forall(t => park.authorizedPets.contains(t)))
  }

people.groupMap(p => s"${p.firstName} ${p.lastName}")(p => parks.filterFor(p).map(_.name))
```

```clojure
(println "10. What are the parks in which each person can walk with all their pets ?")
(defn pets-type [person] (->> person :pets (mapv :type) set))
(defn can-walk [person park] (every? (:authorized-pets park) (pets-type person)))
(let [m (group-by full-name people)]
(apply merge
(map (fn [[k v]] {k (map :name (filter #(can-walk (first v) %) parks))})
m)))
```

Let's compare solutions

Kotlin

```kotlin
println("11. Function composition - findPersonPets")
// Create a function findPersonPets taking 2 function args and returning a composed function of :
// - a searchPerson function taking a String as arg and returning a Person?
// - a petMapper function taking a Pet as arg and returning a String
fun findPersonByFullName(fullName: String): Person? =
    people.firstOrNull { "${it.firstName} ${it.lastName}" == fullName }

fun findPersonPets(
    searchPerson: (String) -> Person?,
    petMapper: (Pet) -> String
): (String) -> List<String> = { fullName -> searchPerson(fullName)?.pets?.map { petMapper(it) } ?: emptyList() }

val composedFind = findPersonPets({ findPersonByFullName(it) }, { pet -> pet.name })
composedFind("Mary Smith")
```

Scala

```scala
println("11. Function composition - findPersonPets")
// Create a function findPersonPets taking 2 function args and returning a composed function of :
// - a searchPerson function taking a String as arg and returning a Person?
// - a petMapper function taking a Pet as arg and returning a String
def findPersonByFullName(fullName: String): Option[Person] = people.find(p => s"${p.firstName} ${p.lastName}" == fullName)

def findPersonPets(searchPerson: (String) => Option[Person],
                   petMapper: (Pet) => String): (String) => List[String] = {
  fullName =>
    searchPerson(fullName)
      .map(person => person.pets.filter(_.`type` == PetType.CAT).map(petMapper))
      .getOrElse(Nil)
}

val composedFind = findPersonPets({findPersonByFullName(_)}, { p => p.name })
composedFind("Mary Smith")
```

Option.of(“THANK YOU”)