Functional programming fundamentals

- Theoretical part
  - What is functional programming
  - Why functional programming matters `15 min`
- DIY part
  - Basic Scala `10 min`
  - Algebraïc data types `10 min`
  - Pattern matching `10 min`
  - Recursion `10 min`
  - Typeclass `15 min`

What is functional programming

As the name suggests, functions are the primary building block of applications. Functions are so called "first class citizens", which means that functions are like data. They can be passed as parameter and a function can return a function. A function that expects a parameter as a function or returns a function is called a higher-order function.

A function in functional programming are expressions, in constrast to imperative programming language where they consist out of a number of statements which will be executed sequentially. A definition given by Wikipedia: “Expressions is a combination of one or more explicit values, constants, variables, operators and functions that the programming language interprets and computes to produce another value". Also, besides only consisting out of expressions, functions need to be pure.

A function is pure if it is referentially transparant. "An expression e is referentially transparant if, for all programs p, all occurrences of e in p can be replaced by the result of evaluating e without affecting the meaning of p. A function f is pure if the expression f(x) is referentially transparant for all referentially transparant x.”. Read that sentence a couple of time and make sure you understand.

Having a referentially transparant function has a couple of nice advantages. Reasoning about software becomes easier and software is more likely to be correct.

Often it's better to use an example to demonstrate something. To show the advantage that we gain of removing side-effects from our function, we'll use an example of a daily thing we all do. This example originally comes from the book Functional Programming in Scala and shows that functions that contain side-effects do not compose. Consider the following function written in Scala. More details will be explained later, but the intend should be clear. This function will give you a coffee given a CreditCard and a PaymentService. There is an obvious side-effect in the function, which is calling the PaymentService. This will charge the CreditCard for the price of a cup of coffee.

```scala
class Cafe {
  def buyCoffee(cc: CreditCard, p: PaymentService): Coffee = {
    val cup = new Coffee()
    p.charge(cc, cup.price)

    cup
  }
}
```

We can show that functions that contain side-effects do not compose by creating a function that buys an arbitrary n cups of coffee. A function like this can be implemented as following.

```scala
class Cafe {
  def buyCoffees(n: Int, cc: CreditCard, p: PaymentService): List[Coffee] = {
    var coffees: List[Coffee] = List()

    for (a <- 1 to n) {
      // prepend the new coffee to the list
      coffees = buyCoffee(cc, p) :: coffees
    }
  }
}
```

While this implementation is technically correct, it's most likely not the result that we want to achieve. The PaymentService will be charged n times and will return n coffees. Also other problems will arise with this implementation. Assume that the charge method on the PaymentService will send a request that has a chance to fail. What do we do when 1 call to the PaymentService fails?

Functional programming to the rescue. We solve this by making the effect explicit in the type signature. In functional programming, we could implement this as following

```scala
class Cafe {
  def buyCoffee(cc: CreditCard): (Coffee, Charge) = {
    val cup = new Coffee()

    (cup, Charge(cc, cup.price)
  }

  def buyCoffees(cc: CreditCard, n: Int) : (List[Coffee], Charge) = {
    val purchases: List[(Coffe, Charge)] = List.fill(n)(buyCoffee(cc))

    // unzip transforms a list of tuples to tuples of two lists.
    val (coffees, charges) = purchases.unzip
    // We combine the charges by reducing n charges to one charge by accumulating t
    (coffees, charges.reduce((c1,c2) => Charge(c1.cc, c1.amount + c2.amount)))
  }
}

class PaymentService {
  // The IO[Unit] can be read as: When I perform, I'm likely to perform some form o
  def charge(charge: Charge): IO[Unit] = ..... // pay the bill
}
```

We obviously still have the side-effect, but we gained several things by encoding it like this.

- By splitting the side-effect from the business logic, we achieved a piece of code that is very easy to test and is deterministic. We can test buyCoffees by just providing a creditcard and the number of coffee we want and we can verify this by just inspecting the output. This is in contract with side-effectful functions/classes, where functions and classes will depend on the dependencies which can be mocks or stubs in test. This way you basically transforms a non-deterministic function to a deterministic function. Also, our code is nothing more than transforming data which is easier to reason about.
- Our function is now reusable as shown in the example. We can compose our functions since it is doing nothing more than the signature actual tells us.

The next chapter will introduce a couple of benefits of functional programming.

Why functional programming matters

Before we can start using functional programming, it's always useful to look at the pre-claimed benefits and what goal it's trying to achieve. In case of functional programming there are several reasons people claim it's great and they explain the major reasons of the increase in popularity. Three reasons will be described briefly below.

Concurrency & parallelism

The importance of concurrency and parallelism is increasing. Gorden Moore, co-founder of Intel predicted in 1965 that the number of transistors on processors would double every year. A couple of years later he adjusted his claim to a doubling every two years. This claim is called Moores Law.

Neil Thompson gave a populair rephrasing of Moores law: "Computing performance doubles every couple of years.". In figure 1 it's easy to see that the number of transistors is still doubling, but the effects positive effects are stagnating.

For that reason, instead of putting more transistors onto a CPU, computers will get multiple CPUs (cores). This means that the time of free performance increases are gone and it's time that the cores are utilised as intended. This means an increase in importance of concurrency and parallelism.

Concurrency and parallelism is very difficult. When using an imperitive programming language, a common this to do is to create variables with values that can change of time. This contains some problems. Look at the example below:

```
public int x = 1;

void function1() {
  int v1 = x;        // 1
  int v2 = x;        // 2

  x = v1 + v2 + x;   // 3
}

void function2() {
  x = x * 5;         // 4
}
```

The problem is that in a multiprocessor environment it is impossible to reason about this code. Let's assume that the lines are executed in the following order: 1, 2, 3, 4. This means that the result will be 15. But in an multiprocessor environment the lines could also be ordered as following: 1, 2, 4, 3. The result in this case will be 7. This problem is called the read-modify-write problem.

When using functional languages where functions are referential transparant, these problems won't exist, since functions are uneffected by side-effects as time, global variables etc.

Basic Scala

Before you are going to practice some basic functional concepts, it's first time to improve your basic Scala.

Paradigms

Scala is a language that fully-supports object-oriented as well as functional programming.

Some functional constructs make use of some features of object-oriented and visa versa. This becomes clear when looking at the ADT, which is modelled using subtyping. More information about ADT will come later.

Classes / Objects / Case Classes / Traits

In Java, there are two types of construct you can use to build up your application. These are interfaces and classes. Scala supports 4 different types and they all have their specific use cases.

Classes

Classes are very similar as in Java. A class is a correlation between its data and methods that operate on this data. Take a look at the following example.

```
class Person(val name: String, val age: Int) {
  def printAge(): Unit = { // this example is not referentially transparant
    println(this.age)
  }
}
```

This class has two properties: a name and an age. Both a public properties and will be accessible on instances of this class. Removing the val keywords in front add private before val will make them private. This class has one method. you can add methods with the def keyword (or val, the difference will be explained later). In this case it takes no parameters and returns Unit. It is good enough for now to read is as void.

Creating an instance of a class is the same as in Java.

```
val person: Person = new Person("Mauro Palsgraaf", 21)
```

Note the explicit type after the variable. Scala has type-inference, which means that it could infer this type by itself. It's a choice to write it explicitly, but I like to do it for clarity.

Although classes are used very often in Java, you wont use them that much when using functional programming. The others are more suitable for the job.

Objects

Objects in Scala are singleton objects. If you ever tried programming languages as Haskell, Elm or Elixir, then you can compare them to modules.

Everything in an object is not correlated to an instance. You can compare them in Java as a class only consisting of static methods.

Case classes

Case classes are very similar as classes, but have some differences. You can define a case class just the same as a class, but with the keyword case in front.

```
case class Person(val name: String, val age: Int)
```

To create an instance of a case class, you can do exactly the same as a class without the new keyword.

```
Person("Mauro Palsgraaf", 21)
```

One of the differences of a case class comparing to regular classes is that case classes are equal by values, not by reference. Look at the following snippet

```
class Person(val name: String, val age: Int)

val person1 = new Person("Mauro Palsgraaf", 21)
val person2 = new Person("Mauro Palsgraaf", 21)

person1 == person2 // this expression will return false

-------------------------------------------------------

case class Person(name: String, age: Int) {
  def printAge(): Unit = { // this example is not referentially transparant
    println(this.age)
  }
}

val person1 = Person("Mauro Palsgraaf", 21)
val person2 = Person("Mauro Palsgraaf", 21)

person1 == person2 // this expression will return true
```

Traits

A trait is very similar to interfaces in Java.

```scala
trait JsonSerializable {
  def toJson: Json
}
```

This trait defined one method without a body, which classes that implement this trait needs to provide. Note that this is not the way to implement this when using functional programming. In functional programming, typeclasses will be used to encode functionality for types. Typeclasses will be explained later.

Besides defining abstract methods, Scala offers the same functionality as default methods in Java 8, whereas implementations can be provided for methods in a trait by composing the other methods in the trait.

Generics

A function using the def keyword, classes, case classes and traits can all made generic. Generics are encoded in Scala as following

```scala
def identity[A](v: A): A
```

In this case, the identity function can take any type as input and returns a value of that type as well. Multiple generics can be specified within the square brackets separated by a comma. The name of the generic doesn't matter, but it's convention to go from A up to Z.

Nothing subtype

It can be neccesary sometime to have a subtype of everything. Most of the time this will be useful when encoding ADT. More about ADT will comes later

Kinds / Types

Covariance

Algebraïc data types (ADT)

Let's consider the Option ADT.

```scala
sealed trait Option[+A]
case class Some[A](x: A) extends Option[A]
case object None extends Option[Nothing]
```

This ADT covers the possibility of missing values. We either have Some with a value of type A or we have None without a value.

Pattern matching

Pattern matching is a construct for checking a value against a pattern. It is comparable to a switch statement in Java-like languages, but more powerful.
I think working with examples is great, so lets dive right into it.

As example, take a look at the following function which adds two to an Option[Int]. If the value is of type Some then add two to the value. If the value is of type None, return None instead

A function like this will looks like this

```scala
def addTwoToOption(optionInt: Option[Int]): Option[Int] = optionInt match {
  case Some(v) => Some(v + 2)
  case None => None
}
```

Exercise

Create an ADT to represent JSON values. JSON values can either be a String, a number, a key value pair (model this as a Map where the key is a String and the value is another json value) or an array of a json value itself

Create an ADT for a binary tree. A binary tree is a generic type where you either have a left or a right value. When finished, verify your answer at page 10.

Recursion

In this chapter, we are going to use are gained knowledge of ADT and pattern matching and use recursion, which is the way of iterating over a sequence. Although recursion is supported in other paradigms, it's not often used. One of the reason is the absence of tail-call elimination, which will be explained later as it's crucial for functional languages.

A recursive function is a function that has a reference call to itself. Let's take a look at the following ADT, which represents a linked list.

```scala
sealed trait List[+A]
case object Nil extends List[Nothing]
case class Cons[A](value: A, tail: List[A]) extends List[A]
```

This is an recursive ADT. It can be read as following: A list of type A is either Nil (which represents an empty list) or a Cons with a value of type A and a tail (which is a list of type A itself).

Let's say we have the following list

```scala
Cons(1, Cons(2, Cons(3, Nil))) // A list with 3 elements: 1, 2, 3
```

We can write a recursive function to calculate the length of a list.

```scala
def length[A](list: List[A]): Int = list match {
  case Nil => 0
  case Cons(_, tail) => 1 + length(tail)
}
```

This function take a list as input. Whenever the pattern corresponds with Cons, we will return 1 + length(tail). If the value corresponds with Nil it will return a 0.

We can use the substitution model to evaluate this expression.

```
1. length(Cons(1, Cons(2, Cons(3, Nil))))
2. 1 + length(Cons(2, Cons(3, Nil))))
3. 1 + (1 + length(Cons(3, Nil)))
4. 1 + (1 + (1 + 0))
5. 3
```

There are 3 laws a recursive function must obey:

- A recursive algorithm must have a base case.
- A recursive algorithm must change its state and move toward the base case.
- A recursive algorithm must call itself, recursively.

The base case is the condition that stops the recursion. In the case of calculating the end, this would be the pattern match on Nil.

The second law states that the state must change and move towards the base case. Looking at the length function, the tail is used inside the recursive function instead of the entire list.

The last law is obvious, which means that a recursive function must call itself.

Exercises

```
Create a recursive ADT representing a binary tree. Make sure it's generic typed.
```

```
Write a recursive function that calculates the sum of all the items of a binary tre
```

tail-recursion

Tail-recursion is a special form of recursion, which is very important in functional programming. To demonstrate it's important, go run the sum function written before with the following argument: (1 until 10000 toList). Before you continue, what happened?

We got a StackOverflow exception. Does this mean we can't do these kinds of things in functional programming? Ofcourse not, tail-recursion to the rescue!

So what exactly happened when we ran the function with a list from 1 to 10000? Every time you call a function, a stack frame will be put onto the stack. A stack is a data structure where you can put data on, but only the item put on the latest can be taken off it. So every time a function is called, a stack frame is added onto the stack containing information about the function call. In our case, we got a recursive function, which will generate a new stackframe work for every recursive call and this will eventually exceed the limit.

Tail-recursion is a recursive function where the recursive call is the absolute latest thing the function does. Take another look at the sum example:

```
def sum(list: List[Int]): Int = list match {
  case Nil => 0
  case x :: xs => x + sum(xs)
}
```

In this case the function can only complete when in completes the recursive call and afterwards still execute the addition to it's current item. If we use the substitution model we get the following result

```
1. sum(Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))))
2. 1 + sum(Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))))
3. 1 + (2 + sum(Cons(3, Cons(4, Cons(5, Cons(6, Nil))))))
4. 1 + (2 + (3 + sum(Cons(4, Cons(5, Cons(6, Nil))))))
5. 1 + (2 + (3 + (4 + sum(Cons(5, Cons(6, Nil))))))
6. 1 + (2 + (3 + (4 + (5 + sum(Cons(6, Nil))))))
7. 1 + (2 + (3 + (4 + (5 + (6 + sum(Nil))))))
8. 1 + (2 + (3 + (4 + (5 + (6 + 0)))))
9. 21
```

The expression is constantly expanding, where at the end when eventually all the recursive calls are executed, the expression needs to be evaluated as a whole.

We can implement sum in tail form as following

```
def sum(list: List[Int]): Int = {
  def loop(list: List[Int], state: Int) = list match {
    def Nil => state
    def x :: xs => loop(xs, state + x)
  }

  loop(list, 0)
}
```

Hands-on

Typeclass