Functional programming fundamentals

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

Introduction

Welcome to the workshop Functional programming, where you will get in touch with the basics of functional programming. This workshop consists of several parts. First, we will explain what functional programming is. Then we will take a closer look at several reasons why functional programming is increasing in popularity. This will be followed by a hands-on, where you will begin your journey through the basics of Scala and FP. You will play with algebraic data types, pattern matching, recursion and typeclasses. All topics will be explained and there are exercises in between that requires you to put the gained knowledge into practice.

Let's get started right away!

What is functional programming

As you would have expected, functions are the primary building blocks of an application written in FP. This is in contrast to object-oriented programming, where classes are used to construct programs. Functions are so-called "first-class citizens", which means that functions are like data. You can just use them as any other ordinary type like String and Int. You can, for example, pass them as a parameter to another function or have a function that returns another function when called. A function that either receives at least one function as an argument or returns a function is called a higher-order function.

Functions in functional programming are expressions, which is in contrast to imperative programming language where they consist of a number of statements which will be executed sequentially. A definition is given by Wikipedia: “Expressions are a combination of one or more explicit values, constants, variables, operators and functions that the programming language interprets and computes to produce another value". There are slightly different flavors of

functional programming. We will pay attention to purely functional programming. This will restrict functions in a sense that they need to be pure. What does it mean to write a pure function?

A function is pure if it is referentially transparent. "An expression e is referentially transparent 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 transparent for all referentially transparent x.". Read that sentence a couple of time and make sure you understand. Simply said, we can substitute the function with it's predicted result without altering the behavior of the program. The following two functions are not referentially transparent:

```
// This function is not referentially transparent, because if we substitute
// the function call to function1() with "hello world",
// we wouldn't have the println("test"), a.k.a we alter the programs behavior
def function1() = {
  println("test")

  // Scala uses implicit return where the last statement will be returned
  "hello world"
}

// This function is not referentially transparent,
// because the output is unpredictable.
def randomNumber() = {
  Math.random() * 10 ....
}
```

There are multiple ways to break referential transparency. Everything that breaks it is called a side-effect. In the previous examples, println("test") and Math.random are the side-effects and the unpredictable parts of the function.

The following functions are referentially transparent

```
// This function is referentially transparent because we can
// substitute every function call with it's output
// If the function call is identity(3), we can replace it
// with 3 without changing the program.
def identity(a: Int): Int = a // returns it's input argument

// This is referentially transparent because we can substitute
// the function by it's output for every combination of x and y
def add(x: Int, y: Int): Int = x + y

add(5, 7) can be substituted with 12
```

Having functions that are referentially transparent provide some great benefits. Reasoning about software becomes easier and software is more likely to be correct. We will discuss some advantages of functional programming in a bit. First, we are going to take a look at why having referentially transparent functions is important.

This example originally comes from the book Functional Programming in Scala and shows that functions that contain side-effects do not compose and are less reusable. Consider the following function written in Scala. More details later, but the intent 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.

```
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.

```
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? Do we need to add glue code here that will handle this and maybe send another request?

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

```
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 the charges
    (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 of
  // side-effect and return a Unit as result
  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 credit card and the number of coffees 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 a 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 for the increase in popularity. Three reasons will be described briefly below.

Correctness

In October 2017, a paper has been released at the University of California focused on different aspects of programming languages and the impact that these have on bugs and errors in software. They took 4 different aspects of languages into account

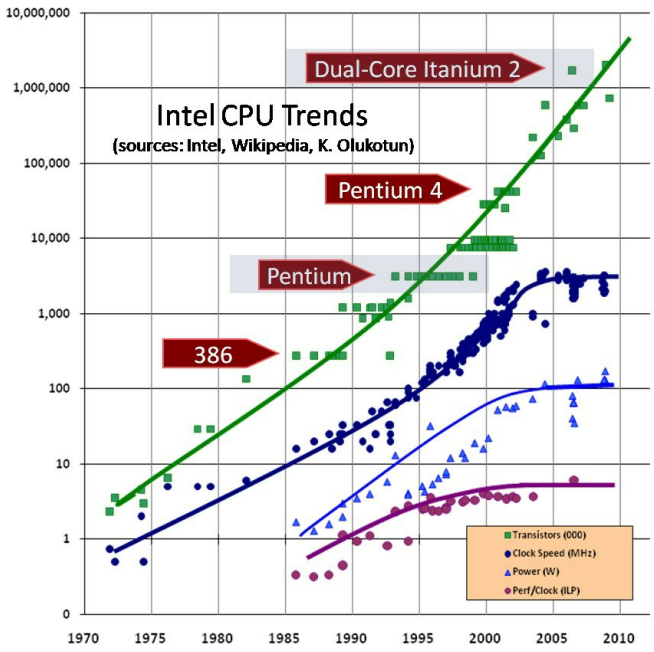
- Functional vs Procedural
- Statically typed vs dynamically typed
- Whether the language allows implicit conversion. (is "7" + 2 allowed for example or is it an error. It could be allowed due to implicit conversion where "7" could be converted to int 7 or 2 could be converted to string "2")
- Managed memory vs unmanaged memory

I really recommend you to read this paper, since the results are absolutely great. It shows that the functional programming languages are by far more correct than procedural languages and that statically typed languages are better in terms of correctness than dynamically typed languages. The paper concludes in the end that statically typed functional programming languages are better than the rest, but I think that is a bit generalized.

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 popular 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 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 utilized as intended. This means an increase in the importance of concurrency and parallelism.

Concurrency and parallelism are very difficult. When using an imperative programming language, a common thing to do is to create variables with values that can change in 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. Assume that function1 and function2 are executed on different threads and that the lines are executed in the following order: 1, 2, 3, 4. This means that the result will be 15. But in a multiprocessor environment, the lines could also be ordered as follows: 1, 2, 4, 3. This will result in 7, since v1 and v2 got the old value, x gets updated and x on line 3 contains the new value. This problem is called the read-modify-write problem and makes reasoning about code

absolutely impossible.

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

Basic Scala

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

Paradigms

Scala is a language that fully supports object-oriented as functional programming. Some functional constructs make use of some features of object-oriented and vice versa. This becomes clear when looking at the ADT, which is modeled using sub-typing. 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 transparent
    println(this.age)
  }
}
```

This class has two properties: a name and an age. The val keyword in front means that you can only get the value, but not set a new one. This can be done by changing it to var. Removing the val keyword will make both properties inaccessible from the outside.

This class has one method. you can add methods with the def keyword. In this case, it takes no parameters and returns Unit. Unit can be read 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.

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.

```
val person: Person = 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)

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.

```
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

```
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 (needs to be capital), but it's convention to going from A to Z.

Nothing subtype

It can be necessary some time to have a subtype of everything. It's the counterpart of Object in Java, where every class extends Object.

Currying

Currying means that we can partially apply a function. In functional languages as Haskell, every function is a one-argument function. For example, a signature that looks like `Int => Int => Int` will take one argument (an Int) and returns a new function with the signature Int => Int. This function takes another Int argument and returns the final Int.

Scala doesn't have only one argument functions. Instead, you can define functions that have multiple parameter lists. For example, the add function. We can apply the add function by specifying the arguments in different argument lists.

```
def add(x: Int)(y: Int): Int = {
    x + y
}

add(5)(7)
```

We can now write a function add7 in terms of add

```
val add7: Int => Int = add(7)

add7(5)
```

This is a very simplistic example, but it's a very important concept in functional programming. If we have a two-argument function and we want to pass it as an argument of a function that expects a one argument function, we can partially apply the function to make it fit. It makes functions more reusable as seen in the example with the add function. We make sure that functions are more reusable because we can write functions in terms of others.

Immutability

Everything in functional programming is immutable. You are not able to change things to something else. For example, adding an item to the end of a list means that we need to create a new list and add the item at the very end.

Why do we embrace immutability in functional programming? First of all, immutability makes it easier to reason about code. Once a value is set, we are sure that it's never going to change. This argument goes hand-in-hand with referential transparency, where changing the value of a variable would be a side-effect in our code. Second, immutable objects are thread-safe which is in contrast with the earlier shown read-modify-write problem in section "Concurrency and

parallelism". We don't have to introduce additional complexity like locks or monitors to make sure our code is thread-safe as we would need when using mutable objects.

A disadvantage of immutability can be memory overhead and performance. If you have for example a list that it going to change often, it needs to construct a list every single time. It has a negative effect on performance since it's slower than just adding a value for example at the end of a mutable ArrayList and it cost more memory since instead of having one structure that is altered many times, you end up with many intermediate structures.

algebraic data types (ADT)

An ADT is a type where we specify the shape of each of the elements. It is encoded using sub-typing. ADTs are used whenever you have a fixed number of things a type can be. They are often used to model our domain, together with type aliases where you give a name to an existing type (for example, type Title = String). A simple example of an ADT is a boolean, it is either true or false. Or a JSON value, which can be a string, object, array, number etc. Let's look at an example:

```
sealed trait Bool
case object True extends Bool
case object False extends Bool
```

We call this ADT Bool. Bool is the so-called Type Constructor. The type constructor is used in the signature, for example as a parameter or a return type.

True and False are data constructors and these are the values that are used in your code. In this case, it's a logical disjunction (or), which specifies that Bool is either a True or a False.

We create a sealed trait. Sealed means that this trait cannot be extended from outside this source file. This is important because we don't want an additional value for Bool.

We can also create an ADT to represent optionality in our code:

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

Again, we create a sealed ADT, because we don't want this trait to be extended outside this file.

[+A] means that this is a generic type, where the A needs to be a type itself, like Option[Int]. It can't, for example, be Option[List] (where the list is sealed trait List[+A]), because List itself is a so-called higher-kinded type instead of a concrete type. You will need to specify the type of the list to actually get a concrete type. This would result in Option[List[Int]] or Option[List[String]].

The + in front means that the value is covariant. It means that if B is a subtype of A than option[B] is a subtype of Option[A].

Exercise

1. Implement an ADT to represent Pets. A Pet can be a cat with a name, a fish with a name and a Color or a squid with a name and age. A Color can be Blue, Green or Orange and can be represented as an ADT as well.

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. It allows us to check a pattern while destructing a value and accessing properties of the pattern.

As an example, we will create a function called not, which will negate the boolean.

```
// note that Bool is the ADT you just created yourself
def not(bool: Bool): Bool = bool match {
    case True => False
    case False => True
}
```

You can see that we match the bool and give it the different options a bool can be. In this case, it's True and False. This is not much different that a switch statement. Let's look at a different example where we use the just created Option ADT for optionality. We create a function addTwoToOption which adds 2 if the option value is present

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

As we can see in the example, it allows us to destruct and get access to values in contrast to

OO-languages like Java where you would implement getters. In our example, the variable v is a wild-card, whereas the pattern matches, will be filled with the value of v. So if Option[Int] is `Some(5)`, v will be 5.

Also, we can stack this. If we have an Option[Option[Int]] we can stack patterns.

```
val optionOption5: Option[Option[Int]] = Some(Some(5))

optionOption5 match {
    case Some(Some(5)) => Do something when both options are some and the value is
    case Some(Some(a)) => Do something when both options are some and the value is
    case Some(None) => Do something when the inner option is not present
    case None => Do something when the outer option is not present
}
```

We see that we pattern match in the first case on a concrete value, but what if we want to match on a predicate function (Int => Boolean for example). We can use guards in that case, where we can extend our pattern match.

```
val x = 3
val optionOption5: Option[Option[Int]] = Some(Some(x))

def isEven(x: Int): Boolean = x % 2 == 0

// y will be 2
val y = optionOption5 match {
  // Do something when both options are some and the value is even
    case Some(Some(a)) if (isEven(a)) => 1
  // Do something when both options are some and the value is odd
  case Some(Some(a)) => 2
  // Do something when the inner option is not present
    case Some(None) => 3
  // Do something when the outer option is not present
    case None => 4
}
```

Remember the sealed keyword when creating ADTs? Whenever we define a sealed ADT, the compiler will help us with pattern matching by doing exhaustivity checks. This means that the compiler verifies that we have a match for all the possible cases.

Exercise

1. create a function `shout` that retrieves a Pet as an argument and returns a String.
   If its a cat, return "Meooowww, I'm $name" where name is the name of the cat.

If its a fish, return "Blubblub, I'm a $color fishy with the name $name".
If it's a squid, return "Hello, I'm $name and i'm $age years old"

Recursion

In this chapter, we are going to use our 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. Sequences as Lists and Trees are a perfect use case for recursion. Lists in functional languages are linked lists. They have a great structure for recursion, where they are modeled as a value and the tail until the end.

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

This is a recursive ADT. It can be read as follows: 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[A] itself).

With this ADT we are able to represent a list

```
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. As you can see in the implementation of length, it calls itself in the body of the function.

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

This function takes 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.

In functional programming, we can use the substitution model where we evaluate the

expressions to values. We can do this because no functions have side-effects. The same input returns the same output and can be substituted by it's expected results.

We can use the substitution model to evaluate this expression.

```
Expression:
length(Cons(1, Cons(2, Cons(3, Nil))))

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
```

Recursive functions have some laws that they 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 the length function, the base case is evaluating to 0 whenever the list is empty. This will stop the recursion.

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 list becomes smaller and gets closer to the empty list.

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

Practice, practice, and practice even more!

Exercises

Note beforehand. A higher-order function (abbreviated as HOF) is a function that takes a function as a parameter or returns a function. Functions in functional languages are just like data, where you can pass them around as arguments to functions or have a function that returns a function.

- Write a function that takes a List[Int] and returns the sum value. Use the following signature:

```
def sum(list: List[Int]): Int
```

- Write a function append that adds an element to the end of the list:

```
def append[A](a: A)(list: List[A]): List[A]
```

- Write a function prepend that adds an element in front of the list:

```
def prepend[A](a: A)(list: List[A]): List[A]
```

- Write a function called concat that concatenates 2 lists:

```
def concat[A](l1: List[A], l2: List[A]): List[A]
```

- Write a HOF called map which takes a list and a function that will transform every element of the list by applying the function:

```
def map[A, B](list: List[A])(f: A => B): List[B]
```

- Write a HOF called filter that takes a list and a predicate and returns a list with all elements where the predicate holds:

```
// This is the scala boolean, not the earlier used boolean
def filter[A](list: List[A])(f: A => Boolean): List[A]
```

- Write a HOF called any which verifies that atleast one element holds to the predicate function:

```
def any[A](list: List[A])(f: A => Boolean): Boolean
```

- Write a HOF called all which verifies that all elements hold to the predicate function:

```
def all[A](list: List[A])(f: A => Boolean): Boolean
```

- Write a HOF called flatMap that takes a list and a function of type `A => List[B]` and returns a List[B]. flatMap should "flatten" the lists. it works like this:

```
-- with Scala list
flatMap(List(1, 2, 3))(x => List(x, x)) should evaluate to List(1, 1, 2, 2, 3, 3)
This is different to map, where the result would evaluate to List(List(1, 1), List(

-- with our list
flatMap(Cons(1, Cons(2, Cons(3, Nil))))(x => Cons(x, Cons(x, Nil)))
```

Hint: use the concat function written before.

```
def flatMap[A, B](list: List[A])(f: A => List[B]): List[B]
```

tail-recursion

Tail-recursion is a special form of recursion, which is very important in functional programming. To demonstrate it's importance, run the following function in the REPL. Before you continue, what happened?

```
// This is not the list we used before, this is scalas list. It's easier for demo p
def sum(list: List[Int]): Int = list match {
    case Nil => 0
    case x :: xs => x + sum(xs)
}

val list = 1 to 10000 toList
sum(list)
```

We got a StackOverflow exception. Does this mean we can't iterate over large sequences in functional programming? Of course 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. So every time a function is called, a stack frame is added to the stack containing information about the function call. In our case, we got a recursive function, which will generate a new stack frame

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 Cons(x, xs) => x + sum(xs)
}
```

In this case, the function can only complete when it completes the recursive call and additionally still execute the addition to reduce the expression to a single value. 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. In the previous example, this is step 8 to 9.

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 Cons(x, xs) => loop(xs, state + x)
  }

  loop(list, 0)
}
```

Let's do the same as we did with the previous example. We can use the substitution model to evaluate all the expressions.

```
1. sum(Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil)))))))
2. loop(Cons(1, Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil))))))), 0)
3. loop(Cons(2, Cons(3, Cons(4, Cons(5, Cons(6, Nil))))), 1)
4. loop(Cons(3, Cons(4, Cons(5, Cons(6, Nil)))), 3)
5. loop(Cons(4, Cons(5, Cons(6, Nil))), 6)
6. loop(Cons(5, Cons(6, Nil)), 10)
7. loop(Cons(6, Nil)), 15)
8. loop(Nil, 21)
9. 21
```

As you can see, when we apply the substitution model on a tail recursive function, the function is rewritten instead of expanded every recursive call. We accumulate the state in the recursive function. We can say that in terms of loop, loop(Cons(6, Nil)), 15) === loop(Nil, 21). These are steps 7 and 8 from the evaluation with the substitution model.

Scala compiler

When writing tail-recursive functions, you can let Scala help you. You can probably imagine a situation where you think you wrote a function with tail-recursion, but you made a mistake and the compiler cannot optimize. This is a bug in our code and we don't want that. To prevent this, we can use the @annotation.tailrec annotation above the recursive functions. Whenever you compile your project, when the function with the annotation is not in tail-call or cannot be optimized, the compiler will give you a warning. With SBT (Scala build tool) you can transform this warning to error if you would like. We are not doing that today.

Hands-on

Rewrite some of the functions written earlier to work on a larger number of items by using tail-recursion. It's up to you how much you want to rewrite, but practice is good. Add the annotation to make sure it's in tail position.

Folding

Take a look at the following functions

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

def product(list: List[Int]): Int = list match {
  case Nil => 1
```

```
    case x :: xs => x + product(xs)
  }
```

We can see that the pattern is the same, only 2 things are different.

- The value when the list is empty
- The function that combines the head of the list to the result of the recursive part.

To abstract over this pattern, we can parameterize the things that are different, which will result in the following signature. We call this function foldLeft

```
def foldLeft[A](list: List[A])(z: A)(f: (A, A) => A): A
```

We can even make this function more generic since we can remove the fact that the output type needs to be the same type as the input type. The new signature will look like this

```
def foldLeft[A, B](list: List[A])(z: B)(f: (B, A) => B): B
```

Exercises

1. Try to implement foldLeft and write the sum and product functions in terms of foldLeft.

We just implemented a function that can reduce a list of values to a single value from left to right. We can also implement a function which folds from the right side. This can be very important since there are functions where order matters. Take subtraction for example with List(1, 2, 3). Folding from the left with the first element (1) as Z will result in (-4) where folding from the right with 3 as Z will result in 0.

Try implement foldRight now.

```
def foldRight[A, B](list: List[A])(z: B)(f: (A, B) => B): B
```

Try to give a smart solution, so don't reverse the list at first!

We are not done yet. After we have discovered typeclasses, we can go a step further and abstract to Foldables!

Typeclass

Typeclasses are simply said better interfaces. Interfaces work based on the subtyping system that object-oriënted languages offer. Classes can implement interfaces which then needs to implement the methods on the interface. There is a simple problem with interfaces, which is that we can't implement interfaces on classes that we didn't create. For example, if we use a library where we have a class that fully satisfy the methods in the interface. Since it's not in the hierarchy, we are not able to use it for polymorphism.

In functional programming, data and things that can be done with the data are two separate things. We can use typeclasses to add functionality to data. As an example, we are going to take a look at a typeclass for verifying equality.

```
trait Eq[A] {
  def equal(x: A, y: A): Bool
}
```

We can implement this trait for every type A. For example, we can implement it for a String, Int and an Option[Int] type

```
val stringEq = new Eq[String] {
  def equal(x: String, y: String): Bool = x.equals(y)
}

val intEq = new Eq[Int] {
  def equal(x: Int, y: Int): Bool = x == y
}

val optionIntEq = new Eq[Option[Int]] {
  def equal(x: Option[Int], y: Option[Int]): Bool = (x, y) match {
    case (Some(a), Some(b)) => implicitly[Eq[Int]].equal(a, b) //ignore this implic
    case (None, None) => true
    case _ => false
  }
}
```

Now that we have type class instances available, we are able to test equality of our different types that implement the type class. To test equality, you can just call the function

```
  intEq.equal(3, 4) // false
```

This is nice, but it's very likely that we want to abstract using typeclasses. Later, when we implement the foldable typeclass, you will see why we want to abstract over it. How do we get our type class instance?
Implicits to the rescue! We can tell Scala that we want to implicitly rely on something. This will be verified at compile-time, so no runtime problems at all. We call this implicit evidence. Assume the following dummy setup. We have a new typeclass called empty. The helloWorldIfEmpty takes a parameter a and will implicitly summon the Eq instance and the Empty instance. Both must be in scope, but don't have to be specified explicitly.

```
trait Empty[A] {
    def empty: A
}

def helloWorldIfEmpty(a: A)(implicit eq: Eq[A], empty: Empty[A]): String = {
  if (eq.equal(a, empty.empty)) "Hello World!"
  else "......"
}
```

The implicit keyword can only be in the last parameter list. If you rely on multiple implicits, you can add them to the last argument list as shown in the example.

We need to change 1 more thing to make this code compile. We need to make the instances implicit. You can do this by adding implicit in front of the val or def.

```
implicit val stringEq = new Eq[String]
```

Let's show how this work together

```
implicit val intEq = new Eq[Int] {
  def equal(x: Int, y: Int): Bool = x == y
}

implicit val intEmpty = new Empty[Int] {
  def empty = 0
}

helloWorldIfEmpty(0) // "Hello World"
helloWorldIfEmpty(1) // "......"

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

implicit val stringEq = new Eq[String] {
  def equal(x: String, y: String): Bool = x.equals(y)
}
```

```
implicit val stringEmpty = new Empty[String] {
  def empty = ""
}

helloWorldIfEmpty("yadayada") // "......"
helloWorldIfEmpty("") // "Hello World"
```

What I personally really like about typeclasses is that they are existential. You can add functionality to your types which will really make sense if you use them more often! It also solves the diamond problem in object-oriented languages. I will not explain it here, but just wanted to mention it and if you are interested, you can look this up on google :)

Abstracting the length function using typeclasses

Remember that we wrote the length function when practicing our recursion skills. It looks like this:

```
def length[A](list: List[A]): Int = list match {
    case Nil => 0
    case x :: xs => 1 + length(xs)
}
```

We did write this function with the help of tail-recursion as well to make sure it's also able to compute the length of a really long list.

```
def length[A](list: List[A]): Int = {
    def loop(list: List[A])(acc: Int): Int = list match {
        case Nil => acc
        case x :: xs => loop(xs)(acc + 1)
    }
}
```

While we practiced our recursion skills, we discovered a pattern and factored this out and called it folding, or more specific foldLeft and foldRight. The length function with foldLeft did look like this:

```
def length[A](list: List[A]): Int = foldLeft(list)(0)((b, _) => b + 1)
```

But, we can go even further! We might also want to know the "length" of a tree, better called

the number of nodes of a tree. Isn't it weird that we are not able to compute the length of a tree? To do so, We need to write a new function. Bummer! I wish there was something called.... typeclasses! There is a special typeclass that we can use perfectly suited for our needs. Take another look at our length function written with foldLeft. The problem is the hardcoded type-constructor (in this case List) that we can fold. We should be able to reuse this function for everything that has a foldLeft function, right? Let's look at foldable. It looks like this.

```
trait Foldable[F[_]] {
    def foldLeft[A, B](fa: F[A])(z: B)(f: (B, A) => B): B
    def foldRight[A, B](fa: F[A])(z: B)(f: (A, B) => B): B
}
```

Exercise

1. Try to implement this Foldable typeclass for List and Binary Tree. Assume that they are endoded like this:

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

sealed trait Tree[+A]
case class Node[A](left: Tree[A], right: Tree[A]) extends Tree[A]
case class Leaf[A](v: A) extends Tree[A]
```

1. Try to make the length function work for both the List and Tree