Übung: Scala Syntax

Einführung in die Funktionale Programmierung

Prof. Frank Puppe, Felix Herrmann, Alexander Gehrke

Sommersemester 2020

Lehrstuhl für Informatik VI, Uni Würzburg

Basics

Before we begin

This introduction compares some concepts with Java, but knowledge in another statically typed language should also be sufficient.

We use Scala 3, the current version of the language. Scala 2 is still widely used, but for functional programming, version 3 has some big improvements, which we heavily rely on.

For a more in depth introduction to the language, refer to the official documentation at `https://docs.scala-lang.org/scala3/book/introduction.html` or the book "Programming in Scala" (Fifth Edition, earlier ones are for Scala 2). In this tutorial, we only explain the parts we use in the lecture.

Required software

For trying out Scala without installing, you can use `https://scastie.scala-lang.org/` (make sure, Scala 3 is selected in the settings).

But for anything larger than single file exercises (e.g. the grade bonus exercises), you should install Scala on your computer. There are different ways to do that and different editors you can use, we will recommend some of them.

Required Software - Command line tools

To install the command line tools, including the build tool `sbt` and the Scala REPL (interactive Scala console), the easiest way is to use Coursier:

`https://get-coursier.io/docs/cli-installation`

Coursier also makes sure you have a JVM installed (Scala compiles to Java Bytecode by default).

Required Software - Editors

A language server named Metals exists for Scala, so you can get completion etc. in any Editor supporting LSP. The one recommended by the developers is VSCode, you can find the plugin here: `https://marketplace.visualstudio.com/items?itemName=scalameta.metals`

After installing, you can open a directory containing a `build.sbt` project definition (a template is provided in the course materials). Select *Import build* when prompted.

For other editors, see `https://scalameta.org/metals`.

Required Software - Using IntelliJ

If you prefer Intellij IDEA, you can install the Scala plugin from the plugin marketplace.

Note that Intellij doesn't use Metals, but implements its own Scala syntax parsing. This may in some cases lead to code being marked as wrong, even if the code compiles, or the other way round.

To open a project, open its `build.sbt` and select "Open as project" when prompted.

Scala by Example

```scala
// comments

/* multiline
 * comments */

/** documentation comments */
val x = 42

def abs(n: Int): Int =
  if n < 0 then -n
  else n

def formatAbs(x: Int) =
  val msg = "The absolute value of %d is %d"
  msg.format(x, abs(x))

@main def myProgram: Unit =
  println(formatAbs(-42))
```

Let’s look at all the parts of this example.

Defining values

```
val x = 42
```

- Values can be declared with `val`. Values are like variables in *math*: they are defined once and have a single value for a given scope[1]
- Type optional, inferred by compiler if not given (still static)
  Type is written between name and equals sign: `val x: Int = ...`
- Mutable variables can be declared with `var` (avoid in FP. Our compiler settings for the exercises will disallow it)

[1]like `final` variables in Java

Function definition

```
def abs(n: Int): Int =
```

- Function defined with keyword `def`
- types for parameters separated by colon, after parameter name
- return type of function after parameter list
- body of function is separated with equal sign and is an expression, i.e. some code that results in a value.
-

Expressions

```scala
def abs(n: Int): Int =
  if n < 0 then -n
  else n
```

- In Scala, nearly everything [2] is an expression, i.e. results in a value, including control structures like `if`.
- `if` evaluates to the result of the taken branch. Each branch is itself an expression.
- last expression of the function is also the result of the function. No `return` statement needed.

[2] Rule of thumb:
if it doesn't introduce any names into the namespace (like declaring a new type or variable), it's an expression (there are exceptions to this, but none that we need during this lecture).

Significant indentation

```scala
def abs(n: Int): Int =
  if n < 0 then -n
  else n
```

```scala
def formatAbs(x: Int) =
  val msg = "The absolute value of %d is %d"
  msg.format(x, abs(x))
```

- In Scala 3, indentation is significant. Here, both lines below each `def` are part of its body.
- Same goes for control structures, e.g. to have multiple lines in an `if`, indent them to the same depth.
- The result of a block with multiple lines is the result of the last expression in that block, e.g. in `formatAbs` the result of `msg.format(...)`

Function type inference

```
def formatAbs(x: Int) =
  val msg = "The absolute value of %d is %d"
  msg.format(x, abs(x))
```

- Return type of functions can be inferred, the compiler decides based on the expression in the function body (use with caution). ⇒ result is the last

  expression in the body
- `msg.format` returns `String`
  ⇒ body evaluates to `String`
  ⇒ return type of `formatAbs` inferred to `String`
- Caveat: compiler won't check, if you return the right thing. Avoid return type inference, especially for larger functions

Main functions

```scala
@main def myProgram(input: Int): Unit =
  println(formatAbs(input))
```

- Main functions are where our program starts (like in C, Java, …)
- Add `@main` before `def` to create a main function
- Main functions may take parameters (only basic types by default, e.g. strings and ints), which can be passed when calling the program

Literals

Literals for basic types are mostly like in other C-like languages:

Integers	`1`, `-1`, Long: `12345678900L`, Hex: `0xF4`
Doubles	`1.00`, `-2.609e11`, Single-precision floats: `0.1f`
Strings	`"Hello, World"`
Booleans	`true`, `false`

String interpolation can be enabled by prefixing a string literal with an `s`. This allows to include values via `$valname` and arbitrary expressions in the string via `${expr}`:

```
val interp = s"Hello, $name. You have ${messages.size} messages"
```

Packages and Classes

Packages and Imports

Packages work like in Java. You add all things in a file to a package by adding a package line like `package my.package.name` at the top. If you come from other languages, the equivalent is often called a module or namespace.

`import` works mostly like in Java. Some differences:

- „static“ imports don’t use any additional keyword
  e.g. `import scala.math.*` allows using the `abs` function without prefix
- To import multiple elements from a package, use braces:

```
import my.package.{SomeClass, myFunction}
```

- For advanced features refer to *Programming in Scala, Odersky et. al., Ch. 12*

Classes

```
class Foo(val field: Int, internal: Boolean):
                   primary constructor

  // functions and vals here
```

- Classes have a primary constructor:
    - parameters in parentheses after class name
    - each param has a name and type
    - params only visible in class body by default. Add `val`/`var` to make them public fields
    - everything in class body (not in a `def`) is evaluated in constructor
- body can contain further instance function and value declarations
- to instantiate, use class name + parameters, e.g.:

  `val myFoo = Foo(7, true)`

Case classes

```scala
case class Book(title: String, author: String, isbn: Long)
```

- Case classes simplify defining value types[3]
- All constructor parameters $\to$ public immutable properties (like declaring with `val` in normal class)
- Can be *pattern matched* (later in this lesson)
- Implicitly declares:
  - `hashCode`, `equals`, `toString`
  - several utility functions

With purely functional code, most if not all your classes will be case classes.

[3]like Java `record` or Python `dataclass`

Objects

```scala
object MyModule:
  def foo(a: Int): Int = ???
  val x = 3
```

- `object` creates singleton types (like a class, which only has one fixed instance)
- Can be used to group functions and values together
- An object defined this way behaves just like objects created from classes
- e.g. calling a function on the object: `MyModule.foo(5)`
- You may define an `object` with the same name as a class. This is called the companion object of a class. Beside grouping standalone functions related to the class, it will be relevant when we learn about typeclasses.

Everything is objects

- In Scala every value is an object (even primitives like `Int`)
- Calling the functions from `MyModule` requires writing e.g. `MyModule.abs(-42)` as it is a function call on object `MyModule`
- Calling functions on literals is possible: `1.toString == "1"`

Traits

- A `trait` is similar to an interface in Java (and compiles to one)
- Most differences (e.g. can have state, different inheritance rules) not relevant to this lecture
- Syntax for abstract functions: simply leave off everything starting at equals sign
- We will use traits for *typeclasses*, which will be explained in a later lecture

Enums

In Scala, `enum` can define an enumeration like in Java, but can also be used for *algebraic data types* (ADT).

- enumeration: set of fixed values, that the enum type can have, e.g.

```
enum Color:
  case Red, Green, Blue
```

- each case is a value of type `Color`:

```
val myColor: Color = Color.Red
```

Enums

- ADT: set of case classes and objects, i.e. at least one of the enum cases has parameters.

```
enum BottleContent:
  case Juice(fruit: String)
  case Water(sparkling: Boolean)
  case Empty
```

- enum cases with parameters are used like case classes:

```
val appleJuice: BottleContent = BottleContent.Juice("apple")
val tapWater: BottleContent = BottleContent.Water(false)
```

- Enums define all their subtypes (you can't add any later on). So we know, that a variable of type `BottleContent` is always either an instance of `Juice` or `Water`, or the `Empty` object.

Inheritance

Class inheritance is rarely used in FP, we'll use typeclasses instead in later lectures. You won't need the following often.

```
class Foo extends Bar with Bax with Qux
```

- (case) classes and objects can extend one class and any number of traits
- if extending a class, it must come first in `extends` clause
- traits can only extend traits

Operators in Scala

- Operators are just “syntactic sugar” for functions in Scala
  e.g `1 + 2` is the same as `1.+(2)`
  (calls a function `+` on the object `1` with parameter 2)
- Any function with a single parameter can be written using this infix notation (no dot and parens)
- For functions with non-symbolic names, use sparingly (future Scala versions will require function definition with `infix` keyword)

Pattern matching

Patterns

Case classes and objects can be used as patterns.

A pattern looks like calling the case class constructor, but may use unbound variables.

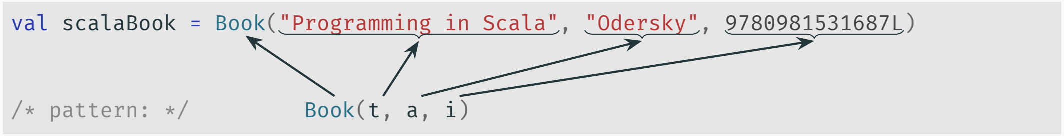
```
val scalaBook = Book("Programming in Scala", "Odersky", 9780981531687L)


/* pattern: */          Book(t, a, i)
```

Patterns

Case classes and objects can be used as patterns.

A pattern looks like calling the case class constructor, but may use unbound variables.



```
val scalaBook = Book("Programming in Scala", "Odersky", 9780981531687L)


/* pattern: */          Book(t, a, i)
```

The variables `t`, `a` and `i` are unbound in the pattern. Matching `scalaBook` and the pattern, these variables are bound to the fields `title`, `author` and `book` in the book class.

Pattern matching

Patterns can be used in a `match` expression:

```
<var> match {
  case <pattern> => <expression using pattern vars>
  case <other pattern> => ...
}
```

The first matched case is evaluated, its value is the value of the match expression. If no case matches, an error is thrown.

When matching on an `enum`, the compiler can warn on non-matched possibilities in most cases (exhaustiveness check).

Pattern matching

A literal, e.g. a string, may also be used as a pattern. This is especially useful with nesting patterns, e.g. `Book(t, "Odersky", i)` to match only if the matched book has the author `"Odersky"`.

Using `_` instead of a variable or constant, a part of the pattern can be ignored (matches anything, does not bind to a name).

An example using all these features:

```scala
scalaBook match
  case Book(title, "Odersky", _) => s"Book ${title} is by Odersky"
  case Book(title, _, _)         => s"Book ${title} is by another author"
  case _                         => "Not a book"
```

Enums and matching

We've heard about enums earlier. These can help the compiler to check, if we cover all cases. Here's our enum from before:

```
enum BottleContent:
  case Juice(fruit: String)
  case Water(sparkling: Boolean)
  case Empty
```

Remember that `BottleContent`'s only subtypes are the cases defined inside.

Enums and matching

Let's use our enum in a match.

```scala
val bottle: BottleContent = ???
bottle match
  case BottleContent.Juice(fruit) => s"You've got $fruit juice"
  case BottleContent.Water(isSparkling) =>
    val variant = if isSparkling then "sparkling" else "still"
    s"You've got water, it is $variant"
```

The compiler will warn us, that we forgot to check for empty bottles:

```
[warn] -- [E029] Pattern Match Exhaustivity Warning: .../code/Enum.scala:24:2
[warn] 24 |  bottle match
[warn]    |  ^^^^^^
[warn]    |  match may not be exhaustive.
[warn]    |
[warn]    |  It would fail on pattern case: Empty
```

Enums and matching

Matching all cases will make the compiler happy:

```scala
bottle match
  case BottleContent.Juice(fruit) => s"You've got $fruit juice"
  case BottleContent.Water(isSparkling) =>
    val variant = if isSparkling then "sparkling" else "still"
    s"You've got water, it is $variant"
  case BottleContent.Empty => "Your bottle is empty."
```

Scala’s Type System

Types

Let's look at Scala's type system, and especially parts that are relevant to FP. Some basic points:

- Primitive types treated like objects syntactically (but are still value types)
- These are named like Java counterpart with uppercase initial letter (e.g. Java `int` $\rightarrow$ Scala `Int`)
- Upper bound for all types: `Any` (includes primitives)
- Upper bound for reference types: `AnyRef` ($\triangleq$ `Object`)

Types: Tuples

- Several values can be grouped into a tuple by enclosing in parens
- Type of tuple is combination of type of parts

```
val myTuple: (String, Int, Boolean) = ("hello", 42, true)
// access members by underscore + 1-based index
myTuple._1 == "hello"
myTuple._2 == 42
myTuple._3 == true
```

- Common usages: intermediate results in chained calls, returning multiple values from functions
- Tuples are immutable

Types: Unit

We've seen functions with type `Unit` used like `void` in Java. But every function in Scala returns a value, so there must be a difference.

- A *unit type* is a type with only one value.
- `Unit` commonly regarded as a 0-tuple
- Value of type unit can be written as `()`

Types: Nothing

- `Nothing` is a subtype of every other type (known as *bottom type*)
- As no object can be of all types at once, there is no value of type `Nothing`
- A function with this type does not return (i.e. has to throw an exception, loop forever, etc.)
- Most useful in combination with type parameters, e.g. an empty list can have element type `Nothing`. We'll see why this is useful, when we look at variance.

Types: Functions

One core concept used in functional programming are Higher Order Functions (HOFs). They are functions that takes another function as parameter.

For this, we need to be able to declare parameters with function type, and ideally function literals (lambdas).

Types: Functions

```
val f1: Int => Boolean = i => i < 3
```

- Function type denoted by `(param types) => return type`
- Function literal follows this syntax, `(params) => expression`

```
val f2: (Int,Int) => Int = (a,b) => a * b
```

- For multiple params, use parens

- Like in Java, single function trait can also be instantiated with function literal

Types: Functions

```scala
val f3: Int => Boolean = _ < 3

val f4: (Int,Int) => Int = _ * _
```

- If parameters are only referenced once in function body, shorthand notation can be used
- Each underscore corresponds to one parameter, in order of occurrence
- Can cause unexpected compile errors when there is nesting involved. If in doubt, use named parameters for more complex lambdas.

Types: Functions — Applying

Functions saved in a `val` can be used just like functions defined with `def`:

```
val f1: Int => Boolean = i => i < 3
```

```
if f1(45) then //...
```

```
val f2: (Int,Int) => Int = (a,b) => a * b
```

```
val i: Int = f2(2,3)
```

Type parameters

Scala supports type parameters for types and functions. These are written in brackets after the type's or function's name:

```scala
trait Container[A]: // ...
```

```scala
case class Pair[A, B](first: A, second: B)
```

```scala
def applyFunction[A, B](input: A, f: A => B): B = f(input)
```

Parameters can be filled with concrete types, e.g.:

```scala
def handle(input: Pair[Int, String]): String = input.second
```

Type parameter inference

Type parameters can also be explicitly given when instantiating a parameterized type or calling a parameterized function, but can be inferred most of the time:

```scala
case class Pair[A, B](first: A, second: B)
```

```scala
val strAndInt = Pair("Hi", 123) // A is String and B is Int
```

```scala
def applyFunction[A, B](input: A, f: A => B): B = f(input)
```

```scala
applyFunction(123, i => i.toDouble) // A is Int, B is Double
// or even
applyFunction(123, _.toDouble)
```

Higher order functions and multiple parameter lists

Function type parameters are usually put into a separate parameter list, as this makes passing multiline functions easier[4]: parameter list with a single parameter can be written in braces to allow line breaks.

```scala
def applyFunction[A, B](input: A)(f: A => B): B = f(input)

applyFunction(123) { i =>
  val half = i / 2
  half.toString
}
```

This is a *curried* function, if you call it with just the parameters of the first list, it will return a function taking the remaining parameters.

[4] in older Scala, they were also needed for type inference

Advanced features

Advanced features

This tutorial only handles concepts we will need in the first few lectures. Some advanced features of Scala will be explained in the lecture, when we need them. This includes:

- *Variance*, which specifies how subtyping for parameterized types works
- *Lazy evaluation*, to compute things only when needed
- *Extension functions* for adding functionality to existing types
- *Givens*, a feature for handling function dependencies without passing them around manually
- *Higher kinded types*, type parameters that take parameters themselves