

## Scala 3 in 6CCS3CFL

For the coursework in this module you are free to use any programming language you like, but I will show you all my code using Scala—I hope you have fond memories of Scala from PEP. The only difference with PEP is that I will use the Ammonite REPL for Scala 3.

If you intend to submit your code for the CW in Scala, you MUST submit code that is compatible with Scala 3!! This is to make it easier for me to test your code and the changes between Scala 2 from last year PEP to Scala 3 in CFL are not that great. In fact, most changes are just some new syntax.

If you need a reminder of the Scala handouts from PEP updated to Scala 3 have a look here [🔗](#). But as said, you do not need to use Scala for the CWs.<sup>1</sup>

The other difference between the Scala I showed you in PEP is that in CFL I will use the Ammonite REPL (with underlying Scala Version 3):

<https://ammonite.io/#Ammonite-REPL>

This is a drop-in replacement for the original Scala REPL and works very similarly, for example

```
$ amm
Loading...
Welcome to the Ammonite Repl 3.0.2 (Scala 3.3.5 Java 21.0.8)
@ 1 + 2
res0: Int = 3
```

Ammonite uses the same Scala compiler, just adds some useful features on top of it. It is quite main-stream in the Scala community and it should therefore be very easy for you to install amm. If you work under a Unix-like system, a sure way to install the right version of Ammonite is by using curl:

```
$ curl -L https://github.com/com-lihaoyi/Ammonite/releases/\
download/3.0.2/3.5-3.0.2 --output amm
```

This creates a file amm which before it can be run might need some adjustments of the permissions. Under recent versions of Windows also have curl, but need a slightly different call:

```
$ curl -L https://github.com/com-lihaoyi/Ammonite/releases/\
download/3.0.2/3.5-3.0.2 --output amm.bat
```

---

© Christian Urban, King's College London, 2020, 2021, 2023, 2025

<sup>1</sup>Haskell, Rust, Ocaml were other languages that have been used previously in CFL. I do not recommend to use Java or C or C++ for writing a compiler, but if you insist, feel free. It has been done before.

Then you need to run Ammonite with `.\amm` and there is no need to change any permissions under Windows.

The big advantage of Ammonite is that it comes with some additional libraries already built-in and also allows one to easily break up code into smaller modules. For example reading and writing files in Ammonite can be achieved with

```
scala> import os._
scala> write.over(pwd / "file.name", "foo bar")
scala> read(pwd / "file.name")
res1: String = "foo bar"
```

The second line writes the string `"foo bar"` into the file `"file.name"`, which is located in the current working directory (`pwd`). We want to implement a compiler—therefore reading and writing files will come in handy.

For loading and accessing code from another Scala file, you can import the code into Ammonite as follows:

```
import $file.name-of-the-file
import name-of-the-file._
```

This assumes the other Scala file is called `name-of-the-file.sc` and requires the file to be in the same directory where `amm` is working in. Again this will be very convenient for our compiler we implement in CFL, because it allows us to easily break up the code into the lexer, parser and code generator.

Another handy feature of Ammonite is that you can mark functions as `@main`. For example

```
@main
def foo() = ...
```

This means you can now call that function from the command line like

```
$ amm file.sc foo
```

If you want to specify an argument on the commandline, say an int and a string, then you can write

```
@main
def bar(i: Int, s: String) = ...
```

and then call

```
$ amm file.sc 42 foobar
```

What is also good in Ammonite is that you can specify more than one function to be “main” and then specify on the command line which function you want to run as entry-point.

Another feature you might like to use is that Ammonite can “watch” files. This means it can automatically re-run a file when it is saved. For this you have to call `amm` with the option `-w`, as in

```
$ amm -w file.sc
```

Of course this requires that you use `println` for inspecting any data as otherwise nothing will be displayed at the commandline.

To sum up, Ammonite is a really useful addition to the Scala ecosystem. You can find more information about how to use it in the first five chapters of the “Hands-on Scala” book by Li Haoyi. These chapters are free and can be used as a reference, see

<https://www.handsonscala.com/part-i-introduction-to-scala.html>

## Some Updates in Scala 3 and the Videos

While Scala 2 and Scala 3 code is on the whole quite compatible, there are some corners where my Scala 3 code differs from the code shown in the videos. I am still fond of using `{...}` rather than Pythonesque indentation syntax. But I switched to the `enum`-syntax for abstract datatypes. Defining regular expressions in the “old” way can be done using abstract classes, like:

```
abstract class Rexp
case object ZERO extends Rexp
case object ONE extends Rexp
case class CHAR(c: Char) extends Rexp
case class ALT(r1: Rexp, r2: Rexp) extends Rexp
case class SEQ(r1: Rexp, r2: Rexp) extends Rexp
case class STAR(r: Rexp) extends Rexp
```

While this code still works in Scala 3 as expected, the definition can now be simplified to:

```
enum Rexp {
  case ZERO
  case ONE
  case CHAR(c: Char)
  case ALT(r1: Rexp, r2: Rexp)
  case SEQ(r1: Rexp, r2: Rexp)
  case STAR(r: Rexp)
}
import Rexp._
```

Note that the syntax with `enum` needs an `import`, otherwise you need to refer to the constructors slightly clumsily as in `Rexp.CHAR('a')` and so on.

Also implicits are now defined differently in Scala 3 and need to be split up into `givens` and `extension methods`. If you want to construct regular expressions using strings, for example `STAR("a")`, then you need to declare a `given`-clause:

```
def charlist2rexp(s : List[Char]): Rexp = s match {
  case Nil => ONE
  case c::Nil => CHAR(c)
  case c::s => SEQ(CHAR(c), charlist2rexp(s))
}
```

```
given Conversion[String, Rexp] = (s => charlist2rexp(s.toList))
```

This uses the auxiliary function `charlist2rexp` which translates a string (list of chars) into a regular expression. The “magic” is then installed in the `given`-clause which calls `charlist2rexp` whenever a `Rexp` is expected, but a string is given.

More convenient operator syntax for regular instructions needs to be defined in Scala 3 as *extension method*. For example, the shorthand-syntax for alternatives, sequences and stars needs to be defined as:

```
extension (r: Rexp) {
  def | (s: Rexp) = ALT(r, s)
  def ~ (s: Rexp) = SEQ(r, s)
  def % = STAR(r)
}
```

After that you can define regular expressions more conveniently as `"ab" | "bc"`, `"ab" ~ "bc"` or `"ab".%`.

To sum up, it should be easy for you to translate the old syntax that is in some places used in the videos and the newer syntax used in the uploaded files. There are a few additional rough edges in the code for parsers, but I will mention them nearer the time.