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 current stable version of Scala, which at teh time of writing is Scala 3.3.1.

> 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.[1]

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 2.5.9 (Scala 3.2.2 Java 17.0.7)
scala> 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/2.5.9/3.2-2.5.9 --output amm
```

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

[1] 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.

```
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`). 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. This will be very convenient for the 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