CPL: Scala

Thomas Van Strydonck, Bob Reynders

Hello World

Goal: run `cpl/HelloWorld.scala`

We recommend the IntelliJ IDEA as development environment for the exercise sessions and for your course assignment, it is available on the PCs at `/localhost/packages/intellij-idea` and it can be started through by invoking `/localhost/packages/intellij-idea/bin/idea.sh`.

Start IntelliJ as soon as possible and make sure to add the Scala plugin in the 'features plugins' window. Notify me during class if you have issues with this. As a backup, the exercises are very easy and can alternatively be completed in an online REPL such as https://scastie.scala-lang.org/ or https://scalafiddle.io/

Download the sources archive from Toledo to start the exercise. Extract the archive somewhere and 'import project' with IntelliJ. Select the `build.sbt` file to do an SBT-driven import. If SBT is not part of the possible imports, restart IntelliJ, if you still have issues – ask me, don't spend 15 minutes on this.

Once your project is imported try and run the provided HelloWorld application. There are several methods to run: "sbt shell -> type run" or "right click -> run".

Exercise 1: Collection API

Goal: get familiar with string and collection operations.

To have you become a little bit more comfortable with Scala, the first exercise is a set of mini problems regarding the (immutable) collection API. Doing these properly and searching for APIs yourself will help you plenty during the assignment!

Complete these small tests without mutable datastructures or references (e.g. no var!) to get used to the immutable collection API.

Look into the `cpl/CollectionTest.scala` file for a definition of the problems, note that these are written as tests and are in `src/test`.

Again, you can run your tests through sbt: “sbt shell -> type test” or “right click” -> run.

Resources:

- https://docs.scala-lang.org/overviews/collections-2.13/overview.html
- https://www.scala-lang.org/api/current/scala/collection/immutable/index.html

Exercise 2a: OO Exp

Goal: learn how to define and inherit traits and define simple classes in Scala.

In this exercise you will define a small expression language in an object-oriented manner. Start from the file: `cpl/oo/exp.scala`, look at the test `cpl/oo/OOTest.scala` for clarity and uncomment them when you are ready to run.

The language is incredibly simple; expressions evaluate to integers and it only has support for literals and addition.

Resources:

- https://docs.scala-lang.org/tour/classes.html

Exercise 3a: FP Exp

Goal: learn how to model typical Algebraic Data Types through sealed traits and case classes and deconstruct the cases using pattern matching. Make sure you know why you want these case class hierarchies to be sealed. If you don’t know, ask me.

Same as above but in Functional Programming style, start from the file: `cpl/fp/exp.scala` and the test `cpl/fp/FPTest.scala`.

Resources:

- https://docs.scala-lang.org/tour/case-classes.html
- https://docs.scala-lang.org/tour/pattern-matching.html

Exercise 2b & 3b: Expanding OO/FP Exp

Goal: learn the strengths and weaknesses between an OO and FP solution. Write your own tests.

Modify exercise 2 and 3. In the OO example, add a new expression to the language: negation. Negation allows you to negate an expression (i.e., `-5` or `-(1+2)`). Write your own test case for these.

In the FP example, add a new operation. Instead of just having `Exp.eval`, interpret the tree with a `Exp.prettyPrint`. It computes a string representation of the expression, uncomment the tests in `cpl/fp/FPTest.scala`.

Resources:

- https://docs.scala-lang.org/overviews/core/string-interpolation.html

Notice how easy the particular change was. No old code had to be touched in each respective solution (if you did, you did weird, discuss with me or a neighbor).

Vice-versa, give the other expansion some thought. What has to change in the OO solution to support pretty printing? What has to change in the FP solution to support negation? Can you do these extensions without changing or duplicating old code? If not, how would you model your solution so that it allows changes in both directions? Discuss this with me or your neighbor, don't get stuck on trying to find a solution.

In programming language research this problem is coined as the "expression problem". The goal is to define new cases to a datatype and new functions over the datatype without modifying or duplicating existing code while retaining type safety. Google for more solutions to this problem if you're interested!

Resources:

- Very simple Scala-based solution to the expression problem
- https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt
- Independently extensible solutions to the expression problem
- Extensibility for the masses

Exercise 4: Student Beverages

Goal: learn how to write classes containing abstract type members. Reflect on the similarities and differences between type members/abstraction on the one hand, and generics/parametrization on the other.

Take a look at the file `cpl/Beverages.scala`. This file contains an object `BeveragesGeneric` describing humans and the type of beverage they like to consume, implemented in Java-like style using generics. Reimplement the same functionality in the object `BeveragesAbstract`, but this time using a type member instead of a type parameter.

Resources:

- https://docs.scala-lang.org/tour/generic-classes.html
- https://docs.scala-lang.org/tour/for-comprehensions.html

Do you prefer the approach using generics, or the type member approach? What do you think the (dis)advantages of one or the other are, and what about the difference in meaning, if any? Discuss with me or your neighbor!

Resources:

- General write-up on SO
- Martin Odersky, the man himself, on the topic

Exercise 5: Delayed Operations

Goal: learn how to define polymorphic classes and implement your own datastructure that can use for-comprehensions.

In the last exercise you will implement `Delay`. Delay allows you to program with delayed computation. It does not compute values unless it is specifically asked to do so through `eval`. It should be possible to use the Delay class with for-comprehensions.

With delay we can describe computations without executing them, this allows programmers to build up computations and pass these around as first-class values without the fear of triggering an expensive or slow computation. For example, piecing together web services as first-class values:

```
// Retrieve the current temperature through a web service
def getTemperature(): Int = ...

// Convert the temperature from celsius to fahrenheit
def convert(celsius: Int): Int = ...

// describe both tasks
val tempComp = Delay.fromFunction(getTemperature)
val convertComp = (celsius: Int) => Delay.fromFunction(() => convert(celsius))
val temperatureInFahrenheit = tempComp.flatMap(convertComp)
```

`temperatureInFahrenheit` now describes a combination between two webservices. It can be passed safely without actually querying anything and higher-order Delay functions could be used to evaluate this in a different thread, on the main thread, etc.

For-comprehensions are a syntax addition for classes that support the methods `map` and `flatMap`, e.g.:

```
val l1: List[Int] = ...
val l2: List[Int] = ...
val l3: List[(Int, Int)] = for {
  i1 <- l1
  i2 <- l2
} yield (i1, i2)
```

For example:

```
for(x <- c1; y <- c2; z <- c3) yield {...}
```

translates to:

```
c1.flatMap(x => c2.flatMap(y => c3.map(z => {...})))
```

Resources:

- https://docs.scala-lang.org/tour/generic-classes.html
- https://docs.scala-lang.org/tour/for-comprehensions.html
- https://docs.scala-lang.org/tutorials/FAQ/yield.html
- Lazy Evaluation