Concepts of Programming Languages
Lecture Notes: Scala

Stefan Mitsch

School of Computing, DePaul University
smitsch@depaul.edu

Scala Pragmatics

We briefly discuss the major steps of setting up Scala and SBT, as well as first steps in using Scala. In Scala, source code files at the top level may contain only class declarations and singleton object definitions.

```
1 object O:
2   // definition of singleton object O
3
4 class C:
5   // definition of class C
```

Small examples can be directly executed in the Scala REPL, larger examples are best typed in a file and then executed. Scala is based on the Java Intermediate Language. In Scala 2 (no longer available in Scala 3), we can disassemble the compiled code in the REPL. All homework assignments come with test cases that you can use to determine whether you implemented the required functionality. The following SBT commands execute unit tests:

```
1 testOnly fp1tests                  // runs all fp1tests
2 testOnly fp1tests -- -n fp1ex05    // runs the test fp1ex05 from
    the suite fp1tests
3 ~testOnly fp1tests                 // runs tests continuously (
    reruns when any file changes)
```

Make sure to run sbt from the correct directory. Type `dir` (on Windows) or `ls` (on Linux and MacOS) and check that the file `build.sbt` is in the current directory

Scala Introduction

Scala is a *multi-paradigm language*, it integrates functional and object-oriented programming. Syntactically, it borrows from Java, ML, and others. Unlike C and many other languages, it does not directly compile to machine code, but instead compiles to JVM Bytecode. This enables bidirectional interoperation between Scala and Java. Its initial design motivation was to build better Java, for distributed systems and data science. Some prominent examples Twitter and Apache Spark, and there is a local Chicago meetup group on Scala. Besides compiling, Scala also has a REPL like Scheme.

Now we take a brief overview of the main syntactic elements in Scala. Scala has the usual Boolean, numeric and string literals as familiar from Java.

```
false || true
1 + 2
("hello" + " " + "world").length
```

We can use all of Java's library.

```
val dir = java.io.File ("/tmp")
dir.listFiles.filter (f => f.isDirectory && f.getName.startsWith
    ("c"))

// with explicit nulls
val dir = java.io.File ("/tmp")
dir.listFiles.nn.map(_.nn).filter (f => f.isDirectory && f.
    getName.nn.startsWith ("c"))
```

Note 1 (Explicit nulls). Explicit nulls change the Scala type hierarchy. By default, Scala follows Java with **null** being a subtype of all reference types. With explicit nulls enabled, the type hierarchy changes so that Null becomes a subtype of Any and Matchable, but no longer of AnyRef. This means, that we cannot initialize references to **null** unless we explicitly declare a union type, e.g., **val** s**:** String | Null = **null**. The preferred way of resolving a union type is pattern matching. The use of the method nn to resolve explicit nulls in the example above is discouraged, because it again results in null-pointer exceptions at runtime.

Unlike in Java, which distinguishes between primitive and complex datatypes, everything in Scala is an object with methods. For example, the integer value 5**:** Int is an object of type Int and it has conversion methods to Double as well as a toString method. Methods can have symbolic names, which allows us for instance to implement numeric operations 5.+ (6). In order to allow us to write those methods in a natural way as operators, any call to a unary function e1.f(e2) can be written as e1 f e2.

```
5 + 6 // 5.+(6)
```

Scala performs static type checking, but just as in Java, all type checks can be made dynamic by casting. We can use the REPL to print types of expressions. Scala comes with glue code that adapts some of the Java types to the Scala type hierarchy:

- Wraps Java primitive types as Scala value types
- Wraps Java reference types as Scala reference types
- java.lang.Object becomes scala.AnyRef

Mutable and immutable data. Its syntax also differs from Java and C in the way that mutable and immutable data are distinguished. In Java and C, variables are by default mutable and reassignment is ok.

```
int x = 10;          // declare and initialize x
x = 11;              // assignment to x OK
```

In Scala, we need to flag mutable objects as being variable.

```
var x = 10           // declare and initialize x (type Int
    inferred)
x = 11               // assignment to x OK
```

Immutable data require different syntax in all three languages.

```
// Java
final int x = 10;  // declare and initialize x
x = 11;            // assignment to x fails, compile error
```

```
const int x = 10;  // declare and initialize x
x = 11;            // assignment to x fails, compile error
```

```
val x = 10         // declare and initialize x
x = 11             // assignment to x fails, compile error
```

Sequencing. Scala supports expression sequencing similar to C and Scheme, but it resembles the sequencing syntax of C statements. When every expression is on its own line, the semicolons are optional.

```
(e_1, e_2, ..., e_n)     // C expression sequencing
(begin e_1 e_2 ... e_n) // Scheme expression sequencing
{e_1; e_2; ...; e_n}     // Scala compound expression

// semicolon optional, whitespace sensitive
{
  e_1
  e_2
  ...
  e_n
}
```

Methods. For expressing methods, Scala again mixes syntax familiar from functional languages with syntax familiar from imperative languages. Methods require type annotations; the return type can usually be inferred, but it is strongly encouraged to annotate types on public methods for documentation purposes (recursive methods always require type annotations). The body of a method is an expression, the value of the expression is returned (void—Unit—is a type whose only value is Nothing!).

```
def plus (x:Int, y:Int) : Int = x + y

// Scala 3 with significant whitespace
class C:
  val x = 1
  lazy val y = 1
  def z = 1

// Scala 2 (closer to Java syntax)
class C {
  val x = 1
  lazy val y = 1
  def z = 1
}
```

Fields and parameter-less methods differ in the time and number of executing the initializer: fields are always initialized on class instantiation and their value does not change once initialized (**val** is strict); lazy fields are initialized on first access, subsequently their value stays unchanged; parameter-less methods are executed every time (**lazy val** and **def** are non-strict). Fields can additionally be distinguished into mutable and immutable fields (**var** vs. **val**).

Scala includes conditional expressions using **if**–then–**else** notation. Like in Scheme, we can use compound expressions to execute expressions for side-effects before returning the value of the last expression. The syntax for compound expressions resembles C statements, but uses expressions!

Structured data. The most basic way of structuring data in Scala is in terms of tuples, which represent a fixed number of (potentially) heterogeneous items, whereas lists are used to represent a variable number of homogeneous items. Both have mutable and immutable variants.

```
final List<Integer> xs = new List<>();
List<Integer> ys = xs;
xs.add(4); xs.add(5); xs.add(6); // mutating list OK
xs = new List<>();               // reassignment fails
ys = new List<>();               // reassignment OK
```

```
val xs = List(4, 5, 6) // scala.collection.List (mutable: scala
    .collection.mutable.List)
var ys = xs
xs(1) = 7              // mutating list fails
xs = 0 :: xs           // reassignment fails
ys = 0 :: xs           // reassignment OK
```

Arrays and more complicated data structures (sets, maps, trees etc.) are available in `scala.collection` as well as through wrappers of the Java library `java.util`. On tuples and lists (and every other data structure that provides a deconstructor—not to be confused with the meaning of deconstructor in C++!), Scala supports manipulation through pattern matching.

Constructing tuples (immutable) tuples in Scala is straightforward, but requires non-trivial implementation effort in Java:

```
val p : (Int, String) = (5, "hello")
val x : Int = p(0)
```

```
public class Pair<X,Y> {
    final X x;
    final Y y;
    public Pair(X x, Y y) { this.x = x; this.y = y; }
}

Pair<Integer, String> p = new Pair<>(5, "hello");
int x = p.x;
```

Pattern matching. Pattern matching is an elegant way to deconstruct data structures into their elements while simultaneously branching on conditions; it is supported in Scala on tuples and lists.

```
// with pattern matching
def a(p:(Int,Int)) = p match
    case (x,y) => x+y

// with projection
def a(p:(Int,Int)) =
  if p==null then throw MatchError(p)
  val x = p(0)
  val y = p(1)
  x + y
```

The syntax in the example above first lists the element that should be deconstructed into its components (`p` **match**), followed by the different options for deconstructing that element. The code branches into these options and picks the first matching case. When matching a case, the components get bound to variables as listed in the case expression (e.g., **case** `(x,y)` binds the left component of the tuple `p` to variable `x` and the right component to variable `y`). Once bound, the variables can be used to produce a result expression (which can be compound). Patterns—similar to ML, Haskell, Rust, or Swift—can be nested and use wildcard operators to ignore irrelevant elements. When used well, pattern matching often improves readability over extensive conditional expressions.

```
def f (xs: List[(Int,String)]) = xs match
    case Nil          => "List is empty"
    case _::Nil       => "List has one element"
    case _::(x,_)::_ => s"The second int is ${x}"   // string
        interpolation, produces result like explicit
        concatenation "The second int is " + x.toString
```

Note 2 (String interpolation). The code above uses string interpolation to build strings instead of explicitly concatenating strings produced by implicit calls to `toString`.

From Scheme, Scala inherits a `cons` operator, but opts for infix notation: **val** `xs = 1 :: 2 :: 3 ::` Nil. The operator is right-associative, so corresponds to `(define xs (cons 1 (cons 2 (cons 3 ()))))`. There is also similarity between linked lists: `(list 1 2 (+ 1 2))` in Scheme vs. `List(1, 2, 1+2)`. Projection onto list elements is done in Scala, like in many languages, with `head` and `tail` vs. `car` and `cdr` in Scheme.

Recursion. Imperative programming often uses mutable data and iteration with loops, whereas functional programming favors immutable data and recursion. This combination of immutability and recursion makes expressing an idea through decomposition into smaller *descriptive* computational artifacts in terms of local state well-understood (as opposed to the *prescriptive* nature of imperative computing through manipulating states in iterations). To make functional programming efficient at runtime, a programming language needs efficient method calls. Compilers of functional programming languages can implement optimizations that transform the functional style into an imperative style in order to avoid the overhead of function calls (tail recursion, more on this later).