Elements of Programming Languages
Lab: Introduction to Scala [DRAFT]

This laboratory session is intended to introduce you to the basic features of Scala, focusing primarily on the *pure, functional* subset that we will work with for most of the course.

Section 1 contains some warm-up exercises and basic information about how to start the interpreter, load code, and so on. The remaining sections describe Scala features (functions, pairs, lists, and objects) and pose 2-3 **exercise**s each.

Submission instructions Once you have finished the lab exercises, you may hand in your completed `Lab.scala` file and this will receive formative (i.e. not-for-credit) feedback. The due date for this is 4pm on October 7.

1 The very basics

Start the Scala interpreter at a DICE command line:

```
$ scala
Welcome to Scala version 2.11.7
   (OpenJDK 64-Bit Server VM, Java 1.7.0_85).
Type in expressions to have them evaluated.
Type :help for more information.

scala>
```

This starts a *read–eval–print loop* (or REPL). If you type a Scala expression into the prompt and hit `<Enter>`, Scala will read and typecheck the expression, evaluate it, and print the result and its type.

1.1 Hello, world!

The `println` function (as in Java) prints to the terminal:

```
scala> println("Hello, world!")
Hello, world!
```

The Scala just prints the terminal output because `println` doesn't return a value. If you ask Scala to evaluate an expression, then it prints out a name for the result value, its type, and its value:

```
scala> 1
res0: Int = 1
scala> "hello"
res1: String = hello
```

1.2 Basic types and operations

Like most languages, Scala has a variety of basic types and operations (booleans, integers, floating-point numbers, strings, etc.)

Before evaluating the following expressions in the Scala interpreter, try to guess what the result will be:

```
scala> 2 + 2
scala> 1 / 2
scala> 1.0 / 2
scala> 1 / 2.0
scala> 1 / 0
scala> 1.0/0.0
scala> true
scala> true == false
scala> true && false
scala> 1 > 1.0
scala> "12345".length
```

1.3 Basic conversions

All objects in Scala have a `toString` method:

```
scala> 1.toString
scala> 1.0.toString
scala> "Hello World".toString
```

Similarly to in Java, the `toString` method is called when a value is referenced in a place where a `String` is expected:

```
scala> println(1)
scala> println(1.0)
scala> println( 1 + " / " + 2 + " = " + (1 / 2))
```

1.4 Comments

Comments in Scala are similar to those in Java:

```
scala> 1 // comment runs until the end of the line
scala> 1 /* delimited comment */ + /* another delimited comment */ 1
```

1.5 Quitting

You can exit the interpreter by typing `:quit` or `<Control-D>`.

1.6 Loading code from a file

Obtain the file `Lab.scala` from the course web page, place it in your current working directory (where the Scala interpreter is running), and load it into the REPL as follows:

```
scala> :load Lab.scala
```

We will use this file for the rest of the tutorial, and from now on, we will refer to code that you should add to it (or commented-out templates you should uncomment and fill in). Once finished, submit this file.

2 Variables, conditionals and functions

2.1 Variable definitions

A declaration defining a variable named `x` in Scala with value `42` looks like this:

```
val x = 42
```

In this case Scala will automatically infer the type `Int` for `x`. The type can also be specified explicitly:

```
val x: Int = 42
```

Specifying the type can be helpful, for example in tracking down type errors where Scala might infer a different type for the value of the variable than you had in mind. **Style note:** In Scala, it is encouraged to write the colon immediately after the variable, and leave a space after the colon.

Scala variables declared using **`val`** are *immutable*: there is no way to change the value of `x` above once it has been declared. There is another way to define *mutable* variables, which will be covered later on in the course. You do not need to, and should not, use mutable variables until then.

2.2 Conditionals

As we've already seen, Scala supports if–then–else syntax familiar from the C/C++/Java family. In Scala, however, an if–then–else construct is an expression, so it can yield a value (provided that both branches of the conditional do so) and it can appear anywhere that an expression can appear.

```
val x = 42;
println(x + "␣is␣" + (if (x % 2 == 0) { "even" } else { "odd" }))
```

(Here, and elsewhere, we use the symbol `"␣"` to indicate a space inside a string.)

2.3 Function definitions

A declaration of a function called `f` that takes an argument `x` (with type `Int`) and returns the value resulting from evaluating expression `E` of type `String` looks like this:

```
def f(x: Int): String = E
```

For example,

```
def incr(x: Int): Int = x + 1
def double(x: Int): Int = x + x
def square(x: Int): Int = x * x
```

Once defined, a function can be invoked using the syntax `f(arg1,...,argn)`, where `f` is the function name and `arg1,...,argn` are the arguments. (Unlike in some functional languages in the ML/Haskell/F# family, these parentheses are required even if the function has only one argument.)

Functions can (as usual) take several arguments, and call other functions (or recursively call themselves):

```
def factorial(n: Int): Int =
  if (n == 0) { 1 } else { n * factorial(n-1) }

def power(x: Int, n: Int): Int =
  if (n == 0) { 1 } else { x * power(x,n-1) }
```

The type of `factorial` is `Int => Int` and that of `power` is `(Int,Int) => Int`.

For any expression, one can ask Scala to compute and display its type as follows. Missing arguments to a function can be written using the underscore character _.

```
scala> :t factorial(_)
Int => Int

scala> :t power(_,_)
(Int, Int) => Int
```

The above examples all involve functions that return a simple Scala expression. It can be handy to, for example, bind a variable to an intermediate value in the body of a function. This can be done in several equivalent ways.

One way is to add braces {...} around the function body and insert a semicolon after the **val** binding:

```
def factorial1(n: Int): Int = {
  val m = n-1 ; if (n == 0) { 1 } else { n * factorial1(m) }
}
```

Alternatively, we can place the **val** binding on its own line inside the braces:

```
def factorial2(n: Int): Int = {
  val m = n-1;
  if (n == 0) {1} else {n * factorial2(m)}
}
```

In either case, the braces are necessary to let Scala know that the function body contains multiple steps.

Style note The semicolon in `factorial2` is optional; if the sequencing of statements is "sufficiently obvious" from indentation context, Scala will infer a missing semicolon. In Scala it is considered good style to leave out such semicolons and use indentation to decrease visual clutter. However, this *semicolon inference* is one of several aspects of Scala that are optimised for experts, not novices. It doesn't hurt to add semicolons explicitly to indicate sequencing, and doing so may make your intent clearer to the Scala parser, or help with debugging. Therefore, we recommend explicitly writing semicolons between consecutive statements while learning Scala.

In fact, Scala also supports a `return` keyword, so we can write something that looks even more like Java if we want:

```
def factorial3(n: Int): Int = {
  val m = n-1;
  if (n == 0) {
    return 1;
  } else {
    return n * factorial3(m);
  }
}
```

However, the `return` keyword in Scala is a no-op: that is, `return E` has the same effect as just `E` on its own.

Exercise 1. *Define a function* `p: (Int,Int) => Int` *such that* `p(x,y)` *is the value of the polynomial* $x^2 + 2xy + y^3 - 1$ *for the given* x *and* y.

Exercise 2. *Define a function* `sum: Int => Int` *such that* `sum(n)` *is the sum of the numbers* $0, 1, \ldots, n$. *For example,* `sum(10) = 55`.

3 Pairs and Tuples

3.1 Constructing pairs and tuples

Scala provides several built-in types for pairs, triples and so on, similar to Haskell's pair and tuple types.

```
scala> val p = (1,"abc")
p: (Int, String) = (1,abc)

scala> val q = (1,"abc",true)
q: (Int, String, Boolean) = (1,abc,true)
```

3.2 Destructing pairs and tuples

Given a pair `p`, the properties `_1` and `_2` access the first and second components respectively. Triples also provide the `_3` property, and so on.

```
scala> p._1
res3: Int = 1

scala> p._2
res4: String = abc

scala> q._2
res8: String = abc

scala> q._3
res9: Boolean = true
```

Exercise 3. *Write a function `cycle` that takes a triple of integers and "cycles through them", moving the first component to the end and the other two forward, e.g. `cycle(1,2,3) = (2,3,1)`.*

4 Pattern matching

4.1 Match/case

Scala also supports a **`case`** construct, similar to the C/C++/Java family. However, there are significant differences:

```
def nameFromNum(presidentNum: Int): String = presidentNum match {
  case 41 => "George␣H.␣W.␣Bush"
  case 42 => "Bill␣Clinton"
  case 43 => "George␣W.␣Bush"
  case 44 => "Barack␣Obama"
}
println("The␣current␣US␣president␣is:␣" + nameFromNum(44))
```

Scala provides a **`match...case`** construct that is similar to the **`switch...case`** construct available in C/C++/Java, but with the ability to match against types other than primitive types. The previous example illustrates pattern matching on integers; we can also define the reverse mapping from president names to numbers in Scala by matching on strings:

```
def numFromName(presidentName: String): Int =
  presidentName match {
    case "George␣H.␣W.␣Bush" => 41
```

```
    case "Bill␣Clinton" => 42
    case "George␣W.␣Bush" => 43
    case "Barack␣Obama" => 44
  }) + "␣US␣president")
println("Barack␣Obama␣is␣the␣"
         + numFromName("Barack␣Obama") + "th␣US␣president")
```

(In the above example, notice that the integer returned by `numFromName` is automatically converted to a string, so that it can be concatenated with other strings.)

Match/case statements also allow for pattern matching in concert with Scala's *case classes*, which will be covered in the next section (and in greater detail later in the course).

Exercise 4. *The above code uses the suffix* `th` *for all numbers, which is wrong for numbers like 41: in English we write* `41st` *instead of* `41th`*.*

Define a function `suffix: Int => String` *that defines the appropriate suffix to use a number as an "ordinal number". That is,* `n + suffix(n)` *should yield the string* `1st` *for 1,* `2nd` *for 2,* `3rd` *for 3,* `4th` *for 4, and so on for any positive integer. You may find the mod operation* `%` *helpful.*

4.2 Case classes

Functional languages such as Haskell provide built-in *datatypes*, which define a type by giving constructors for different cases. Enumerated types (in Java) are similar but much more restricted.

Scala provides a mechanism called *case classes*. We will not fully cover classes until later in the course, but we will use case classes extensively as a basis for defining abstract syntax trees, interpreters and typechecking, and you will need to become familiar with them. Fortunately, case classes can be used for these purposes without going into the details of Scala's object system, inheritance, and so on.

The following is a simple example:

```
abstract class Colour
case class Red() extends Colour
case class Green() extends Colour
case class Blue() extends Colour
```

This code introduces an abstract class of colours, `Colour` and three case classes. Moreover, these classes each have a unique constructor that takes no arguments, and we can create a `Colour` just by writing `Red()`, `Green()` or `Blue()`.

Once we've evaluated these definitions, we can pattern-match against the class names in match/case statements as follows:

```
def favouriteColour(c: Colour): Bool = c match {
  case Red() => ...
  case Blue() => ...
  case Green() => ...
}
```

Exercise 5. *Fill in the definition of* `favouriteColour` *above; if your favourite colour is not one of the available case classes, add it.*

The `Colour` class is basically the same as an enumerated type (`enum`) in Java; the constructors of the three colour classes do not take any arguments. In general, case classes can have multiple arguments:

```
abstract class Shape
case class Circle(r: Double, x: Double, y: Double) extends Shape
case class Rectangle(llx: Double, lly: Double, w:Double, h:Double)
  extends Shape
```

This code introduces constructors for the two shapes, so that we can write `Circle(3,1,2)` for a circle of radius 3 centred at (1,2), or `Rectangle(10,10,1,2)` for a 1-by-2 rectangle with lower left corner at (10,10). (We consider a plane with x axis increasing upwards and y-axis increasing to the right, so that the origin (0,0) is in the lower left corner.)

Moreover, we can use pattern matching to determine both the type and the arguments of an unknown case class such as `Shape`:

```
  def center(s: Shape): (Double,Double) = s match {
    case Rectangle(llx,lly,w,h) => (llx+w/2, lly+h/2)
    case Circle(r,x,y) => (x,y)
}
```

Exercise 6. *Define a function `boundingBox` that takes a `Shape` and computes the smallest `Rectangle` containing it. (That is, a rectangle's bounding box is itself; a circle's bounding box is the smallest square that covers the circle.)*

Exercise 7. *Define a function `mayOverlap` that takes two `Shape`s and determines whether their bounding boxes overlap. (For fun, you might enjoy writing an exact overlap test, using `mayOverlap` to eliminate the easy cases.)*

Note: It is a little awkward to write **`case class`** `Red()` and write `Red()` in pattern matches. In Scala, one can write **`object`** instead of **`class`** for a class that has only one instance; thus, for case classes with no arguments we can write:

```
case object Red extends Colour
...
c match { case Red => ...}
```

We will see another example of this in the next section where we write **`case object`** instead of **`case class`** for the `Nil` list constructor.

5 Higher-order Functions and Lists

5.1 Anonymous and Higher-Order Functions

Scala also supports *anonymous* functions (i.e. lambda-abstractions / closures) that may be familiar from Haskell.

```
val anonIncr = {x: Int => x+1} // anonymous version of incr
val anonAdd = {x: Int => {y: Int => x + y}}
```

Exercise 8. *Using anonymous functions, define the function `compose` that takes two functions and composes them:*

```
  def compose[A,B,C](f: A => B, g : B => C) = ...
```

The above exercise illustrates another feature of Scala (that may also be familiar from Java and Haskell), namely, type parameters. In the definition of `compose`, we abstract over the types `A`, `B` and `C`. This function can be applied to any functions having compatible result and argument types. Often, Scala's built-in type inference algorithm can work out what the values of `A`, `B` and `C`, but if not, you can provide them directly, e.g. writing `compose[Int,String,Bool]`.

Exercise 9. *Define two expressions `e1`, `e2` such that*

```
   compose[Integer,String,Boolean](e1,e2)
```

typechecks. Evaluate the following expressions in Scala, with the expressions `e1`, `e2` *you have defined, and include the output in your solution in a comment.*

```
compose(e1,e2)
compose(e2,e1)
```

5.2 Lists

Other types can also take parameters; lists exemplify this pattern. In Scala, lists are defined (more or less) as follows, using case classes:

```
abstract class List[A]
case object Nil[A]
case class Cons[A](hd: A, tl: List[A])
```

Here, `[A]` is a type parameter to the `List`, `Nil` and `Cons` types, and indicates what the type of the list elements is. For example, we can form types `List[Integer]` of lists of integers, `List[(Integer,String)]` of lists of pairs of integers and strings, or `List[List[String]]` of lists *of lists* of strings.

Technically, Scala does not actually define a class named `Cons`; instead, it defines a class called `::` so that one can write `x :: l` to construct a list with head `x` and tail `l`, and one can pattern match on lists as follows:

```
def isEmpty[A](l: List[A]) = l match {
  case Nil => true
  case x :: y => false
}
```

Moreover, in Scala, a list can also be constructed as follows:

```
List() // same as Nil
List(1) // same as 1 :: Nil
List(1,2,3) // same as 1 :: 2 :: 3 :: Nil
```

Exercise 10. *Define a function* `map` *that takes a function* `f: A => B` *and a list* `l: List[A]` *and traverses the list applying* `f` *to each element.*

```
def map[A](f: A => B, l: List[A]): List[B] = ...
```

Exercise 11. *Define a function* `filter` *that takes a predicate and a list and traverses the list, retaining only the elements for which* `p` *is true.*

```
def filter[A](p: A => Boolean, l: List[A]): List[A]
```

Exercise 12. *Write a function to reverse a list.*

```
def reverse[A](l: List[A]): List[A] = ...
```

5.3 Using Scala's built-in list operations

Scala provides a number of library operations on `List`s, as methods of the (abstract) `List` class. We haven't covered classes and methods in depth in this tutorial, but they are commonly used in Scala code. For example, the map and filter operations can be called as follows:

```
scala> val l = List(1,2,3,4,5,6)
```

```
l: List[Int] = List(1, 2, 3, 4, 5, 6)
scala> l.map(x => x + 1)
res1: List[Int] = List(2, 3, 4, 5, 6, 7)
scala> l.filter(x => x % 2 == 0)
res2: List[Int] = List(2, 4, 6)
```

6 Maps

Scala has a large library of collection data structures, including various forms of maps. We will mostly use the simplest form of these, `ListMap`, which are essentially lists of key-value pairs. The `ListMap` implementation isn't very efficient, but it is simple and we will mostly use two basic operations that find the value of a key, or create a new key-value binding.

We will first ask you to explicitly define operations for looking up and updating list maps, using the type `List[(K,V)]`, rather than the Scala collection library's `Map` interface. The basic operations on list maps are as follows:

- `empty` is a constant list map containing no elements.
- `lookup(m,k)` returns the value of key `k` in `m`, if any
- `update(m,k,v)` returns a new map extending with key-value pair `(k,v)`
- `keys(m)` returns the list of keys in map `m`

The empty map is represented using the empty list:

```
def empty[K,V]: List[(K,V)] = List()
```

Notice that we need to parameterise the definition by the key and value types. Likewise, we can represent a simple map that associates the first three positive integers with the first three letters of the alphabet as follows:

```
val map123 = List((1,"a"),(2,"b"), (3,"c"))
```

The lookup operation takes a list of pairs and a key, and returns the value associated with the key, if any. For example,

```
scala> lookup(map123, 2)
res1: String = b
```

We assume that there is at most one matching key in the list; the behavior if there is no matching key is undefined.

Exercise 13. *Define the `lookup` function:*

```
def lookup[K,V](m: List[(K,V)], k: K): V = ...
```

The `update` function takes a list map `m`, a key `k`, and a value `v`, and constructs a new list map `m2` such that `m2(k) = v` and `m2(k0) = m(k0)` for any `k0` different from `k`. This function should preserve the invariant that each key has at most one value. For example:

```
scala> update(map123, 2, "asdf")
res1: List[(Int, String)] = List((1,a), (2,asdf), (3,c))
scala> update(map123, 4, "d")
res2: List[(Int, String)] = List((1,a), (2,b), (3,c))
```

Notice that list maps are "immutable": that is, the update function does not change the value of its argument, it just creates a new list map with the new binding.

Exercise 14. *Define the* `update` *function:*

```
def update[K,V](m: List[(K,V)], k: K, v: V): List[(K,V)] = ...
```

Finally, the `keys` operation produces a list of the keys present in a list map. For example:

```
scala> keys([(1,"a"),(2,"b"), (3,"c")])
[1,2,3] : List[Int]
```

Exercise 15. *Define the* `keys` *function.*

```
def keys[K,V](m: List[K,V]): List[K] = ...
```

6.1 Using Scala's built-in `ListMaps`

In Scala, one can create an instance of the `ListMap` class as follows:

```
val map12 = scala.collection.immutable.ListMap(1 -> "a", 2 -> "b")
```

Technically, what this does is create an *immutable* list map that maps 1 to `"a"` and 2 to `"b"`. Immutable means that the value of the map will never change (like in Haskell, but in contrast to typical Java map data structures). Moreover, the types of the keys and values of this map will be inferred automatically.

Exercise 16. *Define the mapping from president numbers to names from Section 4.1 as a value*

```
val presidentListMap: ListMap[Int,String] = ...
```

using Scala's built-in list maps.

In Scala, the `lookup` operation is just written like function application: `m(k)`. For example, using `map12` above, `map12(1)` will evaluate to `"a"` and `map12(2)` will evaluate to `"b"`. Applying `map12` to any other integers will result in an error. Scala's standard library also has a built-in version of the `update` operation, written `m + (k -> v)`.

If you want to create an empty map, the following doesn't quite work:

```
val empty = scala.collection.immutable.ListMap()
```

Scala will infer that they key and value types are a special type `Empty`, which usually isn't what you want. Instead, to create an empty map with specific key type `Int` and value type `String`, do this:

```
val empty = scala.collection.immutable.ListMap[Int,String]()
```

Exercise 17. *Define map* `map12` *using the empty map and Scala's update function.*

Exercise 18. *The Scala* `ListMap` *class provides a method* `toList` *that converts a* `ListMap[K,V]` *to a* `List[(K,V)]`*. Define a function that converts a* `List[(K,V)]` *back into a* `ListMap[K,V]`*.*

```
def list2map(l: List[(K,V)]): ListMap[K,V] = ...
```