Using Scala in CS241

Winter 2018

Contents

1 Purpose

This is a guide to some of the useful features in Scala 1.12.1 which you may find helpful in CS241. It is not intended as a standalone tutorial for Scala. It assumes you are familiar with Racket (CS135, CS136) and C/C++ (CS136, CS246).

2 Scala

Scala is an object-oriented functional programming language which runs on the Java Virtual Machine (JVM). It draws heavily from Java (which in turn draws from C++) and from functional languages such as Scheme (which Racket is a dialect of) and Haskell. Code can be written in either a functional or imperative style, but will generally be cleaner and more readable in a functional style. Any methods and classes from Java are available in Scala, but there are usually Scala equivalents which are more suited to the language.

To use Scala on the `linux.student.cs.uwaterloo.ca` environment you will need to `source /u/cs241/setup` to have access to the necessary tools.

3 Basic Syntax

Scala uses mostly C-like syntax and should look fairly familiar to programmers used to C++ or Java. There are a few notable changes:

1. Types come after names, not before. To specify a definition you should use one of "def" (for a function), "val" (for a constant variable: variables should usually be constant), or "var" (for a non-constant variable: use sparingly).

```
//Java function to compute the factorial of a number using iteration
int fact(int n) {
  int prod = 1;
  while (n > 0) {
    prod = prod * n;
    n = n - 1;
  }
  return prod;
}

//Scala function to compute the factorial of a number using iteration
//Types come after names and function signatures.
//Functions have an ``='' after their definition. This is because
//they don't always need a block afterwards, just any value.
def fact(n: Int): Int = {
  //Parameters are immutable, and we want to mutate n
  var counter = n
  //Prod is a mutable variable of type Int. Scala automatically
  //figures out it should be an Int based on the rest of this function.
  //The braces here do nothing, but illustrate that you can have braces
  //almost anywhere in Scala. Braces work like `begin' in Racket.
  var prod = { 0 }
  //You can write while loops which look the same as those in Java
  while (counter > 0) {
    //Semicolons are not necessary,
    //unless you want multiple statements per line
    prod = prod * counter
    counter = counter - 1
  }
  //The last value in a block is the block's value, similar to `begin'
  //in Racket, so we don't need a ``return'' here.
  prod
}
```

2. Arrays, maps and so forth are indexed with parentheses rather than square brackets. Templates and generics are instantiated with square brackets rather than angle brackets:

```
//Define an array of five integers in Java and access the fourth integer
int[] ints = {1,2,3,4,5};
ints[3];

//The same thing in Scala
val ints = Array[Int](1,2,3,4,5)
ints(3)
```

3. `if` is an expression (similar to the `?:` operator) rather than a statement:

```
//Java if statement
if (x % 2 == 0) {
  return x;
} else {
  return 2;
}

//Java ternary expression
return (x % 2 == 0) ? x : 2

//Scala if expression
if (x % 2 == 0) x else 2
```

4 Tuples, Arrays, Lists and Vectors in Scala

Scala supports a number of different sequence types (lists of data sorted in order), included most of those that you are familiar with. All of them are grouped under a common abstract parent class called `Seq`, which supports appending to the front, appending to the end, sequence concatenation, random access, and so forth, but not necessarily with good performance guarantees for any operation. In general, collections in Scala are immutable. You can use seqs as follows:

```
//The [Int] type here is automatically inferred
val seq1: Seq[Int] = Seq(1,2,3,4,5)
val seq2: Seq[Int] = seq1 :+ 6 //(1,2,3,4,5,6)
val seq3: Seq[Int] = 0 +: seq1 //(0,1,2,3,4,5)
val seq4: Seq[Int] = seq2 ++ Seq(7,8) //(1,2,3,4,5,6,7,8)
val four: Int = seq1(3) //4
val seq5: Seq[Int] = seq1.tail //(2,3,4,5)
val seq6: Seq[Int] = seq1.init //(1,2,3,4)
val one: Int = seq1.head //1
val five: Int = seq1.last //5
```

Sometimes you may wish to make sure that specific operations are efficient. For example, say you want very efficient `tail` and `+:` operations but don't care about the efficiency of other operations. Then as you will recall from Racket, a list would be appropriate.

```
val lst1: List[Int] = List(1,2,3,4,5)
val lst2: List[Int] = lst1.tail //This is O(1)
val seq: Seq[Int] = lst1.toSeq //You can convert lists to sequences
val lst: List[Int] = seq.toList //And sequences to lists
//:: is the 'cons' operation. It's equivalent to +: for seqs, but has a
//different name as a reminder that it's constant time and only works on lists.
val lst3: List[Int] = 0 :: lst1 //(0,1,2,3,4,5)
```

If you want guaranteed efficient random access, you can use an IndexedSeq:

```
val iseq1: IndexedSeq[Int] = IndexedSeq(1,2,3,4,5)
//You can convert IndexedSeqs (or anything else) to sequences
val seq: Seq[Int] = iseq1.toSeq
val iseq: IndexedSeq[Int] = seq.toIndexedSeq //...and back
val three: Int = iseq1(2) //Guaranteed to be O(1)
```

You can find mutable versions of most collections in `scala.collection.mutable`, but you should try to stick to immutable versions whenever it is practical. Sequences support all of the operations that you may have used on Racket lists, and many more besides:

```
val seq1: Seq[Int] = Seq(1,2,3,4,5)
//'x => foo' is an anonymous function
val evens: Seq[Int] = seq1.filter(x => x % 2 == 0) //(2,4)
val timestwo: Seq[Int] = seq1.map(x => x * 2) //(2,4,6,8,10)
//We can deconstruct pairs using curly braces and 'case'.
//The initial value gets passed as a separate argument to foldLeft.
val sum: Int = seq1.foldLeft(0){case (a,b) => a + b}
```

You can also use collections iteratively in the way that you might be used to from Java, Python or similar:

```
//Java code
int[] ints = {1,2,3,4,5};
//Print 1 2 3 4 5 on separate lines
for (int i: ints) {
  System.out.println(i);
}

//Scala equivalent
val ints = Seq(1,2,3,4,5)
for (i <- ints) {
  println(i)
}

//You can also use 'to' or 'until' as a short form:
for (i <- 1 to 10) {
  println(i) //Prints 1 2 3 .. 10
}
for (i <- 1 until 10) {
  println(i) //Prints 1 2 3 .. 9
}
```

Sometimes you just want to pass, return or store several things bundled together. In Java, Racket, C++ etc you might use a list, create a data structure, use a standard data structure such as `std::pair`, etc. In Scala you can just pass them in a tuple:

```
;Racket
(define (return-three-numbers)
  (list 1 2 3))
(second (return-three-numbers)) ;returns 2

//Scala
def returnThreeNumbers(): (Int, Int, Int) = (1,2,3)
returnThreeNumbers()._2 //returns 2
```

5 Binary output in Scala

Scala does not have its own support for binary I/O, but you can use Java's methods to do so.

```
//Print a single byte to stdout
val b: Byte = 48
System.out.write(b) //Prints 0

//Print a bunch of bytes to stdout at once
val bs: Array[Byte] = Array(48,49,50,51,52)
System.out.write(bs) //Prints 01234

//You should make sure to close the output stream
//at the end of your program to ensure no output is lost
System.out.close()
```

6 Maps

Scala supports both immutable and mutable maps, like with any other collection.

```
//An immutable map
val m: Map[String, Int] = Map()
val m2 = m + ("foo" -> 5)
m2("foo") //returns 5
var m3 = m2 //Store our immutable map in a mutable variable
m3 = m3 + ("bar" -> 10) //...so that we can add things to it easily
m3("bar") //returns 10

//If we import mutable data structures, we can also mutate the map directly.
import scala.collection.mutable

val mm = mutable.Map[String,Int]()
mm.update("foo",5)
mm("foo") //5
//Bar is not set, so set bar to 5 and return it
mm.getOrElseUpdate("bar",5)
//Foo is already set, so just return 5 and don't change it
mm.getOrElseUpdate("foo",10)
mm("baz") //Crashes, since baz isn't in the map
mm.get("baz") //Returns None
mm.get("foo") //Returns Some(5)
```

7 Option types

Like in many languages, computations in Scala can fail. In many languages, you can handle failures by either checking them in advance or capturing an exception once the computation fails. Scala prefers to handle things without exceptions by using the "Option" type:

```
val anint: Option[Int] = Some(5) //Contains an int
val notanint: Option[Int] = None //Doesn't contain an int
//Does it contain an int? Nobody knows for sure!
val maybeanint: Option[Int] = if (random()) anint else notanint
```

```
    //If maybeanint actually has an int in it, return it.
    //Otherwise just return 0
    if (optionanint.nonEmpty) maybeanint.get else 0

    anint.get //Returns 5
    notanint.get //Crashes!

    //Same as the if statement above. More on this later.
    maybeanint match {
      case Some(i) => i
      case None => 0
    }
```

8 Objects and Classes

Like in Java, everything in Scala must be in a class. However, to avoid the mess of static methods in cases like `java.lang.Math`, you can also create "objects" directly, which work like singleton classes. As a result, there are no static methods or values in Scala.

```
  //java class with static
  class Foo {
    static void bar() { ... }
  }
  //...later...
  Foo.bar()
  //...

  //scala object
  object Foo {
    def bar(): Unit = ...
  }
  //...later...
  Foo.bar()
  //...

  //Java instantiated class
  class Foo {
    public int member;
    Foo(int i) {
      member = i;
    }
  }
  new Foo(5).member //returns 5

  //Scala instantiated class
  class Foo(i: Int) { //Constructor arguments are private fields listed here
    val member = i //Fields are public by default
  }
  new Foo(5).member //returns 5
```

9 Pattern Matching and Case Classes

You can use pattern matching to make it much easier to look at the contents of lists, tuples, options and other composite types. Pattern matching is done with the `match` statement and looks vaguely similar to a `switch` statement from Java/C++. If you have ever used `match` in Racket, Scala's `match` is very similar.

```
//Java
int i = 5;
switch (i) {
  case 0:
    return 0;
    break;
  case 3:
    return 3;
    break;
  default:
    return 2;
}

//Scala
val i = 5
i match {
  case 0 => 0
  case 3 => 3
  //_ means 'anything' here. Since cases are processed in order,
  //this will handle anything except 0 and 3.
  case _ => 2
}
```

However, pattern matching is much more powerful:

```
val ssi: Seq[Seq[Int]] = Seq(Seq(1,2), Seq(3,4), Seq(), Seq(5,6,7))
ssi match {
  //Do this if ssi was the empty sequence
  case Seq() => 0
  //Do this if ssi was a sequence containing the empty sequence
  case Seq(Seq()) => 1
  //Do this if ssi has two sequences, and put them in variables s1 and s2
  case Seq(s1,s2) => s1.length + s2.length
  //Do this otherwise, but make sure we pull out
  //the first element of ssi into a separate variable
  case first +: rest => first.length + rest.length
}
```

You can also use pattern matching directly in variable definitions:

```
//This works with any 'case class': see below
val seq: Seq[Int] = Seq(1,2,3)
val Seq(i,j,k) = seq //i = 1, j = 2, k = 3

val triple: (Int, Int, Int) = (3,2,1)
val (i,j,k) = triple //i = 3, j = 2, k = 1
```

If you want to use your own classes in pattern matches, you should make them "case classes". This changes a few things, notably:

- The arguments of the constructor are made public fields of the class rather than private fields.
- You no longer need the "new" keyword to instantiate the class.

```
//Skip the body of this class since we don't need it here
case class Tree(data: Int, children: Seq[Tree])

val myTree: Tree = Tree(5,Seq(Tree(3,Seq()),Tree(2,Seq())))

//This confusing syntax uses pattern matching to create a variable
//called ''myTreeData'' which contains the first field of myTree,
//which is 5. Its type is automatically deduced as int.
val Tree(myTreeData, _) = myTree

//A generic version of Tree which works for any data type
case class Tree[T](data: T, children: Seq[Tree[T]])

val myIntTree: Tree[Int] = Tree(5,Seq())
```

10 Further Reading

You will find the Scala documentation helpful in finding more features of the language: this document only scratches the surface. You can find the documentation on the Scala website but you may find it easier to google "Scala Seq" or similar and click on the appropriate link to find the documentation you're looking for.