

4.6

Updated: 2021-03-08

Review: Syntax and Collections

© 2016 Robin Hillyard



Northeastern
University

Part one

Syntax

Scala programs and syntax

- I've explained before about the nature of Scala programs:
 - Basically, a Scala program is an expression that yields a result* but...
 - there are other constructs which you can insert before the expression such as:
 - type definitions (traits, classes, etc.) (including methods and initialization—expressions yielding Unit);
 - method definitions (where the RHS is an expression);
 - val/var definitions (where the RHS is an expression);
 - imports;
 - syntactic sugar such as for-comprehension, case clause.
 - Usually, if you add such things to your expression you will need to create a block with {}

*** So, you don't need to end an expression with "return" since that's what's expected**

Lines and blocks

- Lines:
 - Scala lines don't normally need a semi-colon at the end of each line because they are not normally *statements* (but there are exceptions to this which we will cover later—and which the compiler will warn you about)
 - The compiler pseudo-inserts a semi-colon for you at the end of each line if it thinks it's a statement rather than an expression.
 - So, if you write:
expression1
+expression2
it will be interpreted as a statement followed by an expression
expression1; **and** +expression2
- You can fix this by putting parentheses around your expression or by moving the operator up to the first line:
(expression1
+expression2)
expression1+
expression2

Lines and blocks (contd.)

- Blocks:
 - they allow you to precede your expression by `val/var/def` statements;
 - they reduce namespace conflicts;
 - they allow for encapsulation (information hiding);
 - there is no requirement for method or identifier definitions (`defs`, `vals`, etc.) to be at any particular level: private methods will normally be inside a public method definition (or a class)
- Even import statements can be inside blocks

```
val x = 3
def y = {
  val z = sqr(y)*x
  import math.round
  def sqr(p: Double) = round(p*p).toInt
  z
}
```

Basic Scala syntax*

- `module ::= prolog type-definition*`
- `prolog ::= package import*`
- `type-definition ::= header definition* “}”`
- `header ::= trait identifier mixins “{“ |
 [“abstract”|“case”] “class” identifier type-declaration
parameter-set* mixins “{“ auxiliary-constructor* initialization |
 object identifier “{“`
- `mixins ::= “extends” type [“with” type]*`
- `definition ::= method-definition | variable-definition | type-definition`
- `method-definition ::= “def” identifier [parameter-set]* “:” return-type [“=“ expression]`
- `variable-definition ::= [“lazy”] “val”|“var” identifier “:” return-type [“=“ expression]`
- `expression ::= identifier | invocation | “{“ definition* expression “}”`
- `invocation ::= [receiver] [“.”] identifier [identifier | [“(“ expression* “)”]*`

* For the true syntax see <http://www.scala-lang.org/docu/files/ScalaReference.pdf> p. 159

Parentheses

- Parentheses are generally there to override the precedence rules for expressions. But occasionally, there's a bit more to it.
 - IntelliJ/IDEA and Eclipse have analyzers which will tell you if you have superfluous parentheses.
 - Joke: *what does LISP stand for?* Lots of irritating superfluous parentheses
 - For example, you don't need parentheses around a singleton parameter type of a function type—these are the same:

```
val x_f_t: Try[(X)=>Fitness] = for ((t, s) <- matchFactors(factor, `trait`)) yield fc(t)(s)
```

```
val x_f_t: Try[X=>Fitness] = for ((t, s) <- matchFactors(factor, `trait`)) yield fc(t)(s)
```

- But, it's fairly conventional to put the parentheses there, in parallel so to speak with the function invocation. And if your parameter list is a tuple (“parameter set”), you must use parentheses.
- You will need parentheses here, though:

```
trait Cache[K, V] extends (K => Future[V])
```

Syntax of lambdas

- Lambdas involve pattern-matching so here's a summary of their rules:

- You don't need parentheses for a lambda provided that the context clearly requires a function and the type of the “_” can be inferred:

```
scala> def doubleMap(f: Int=>Int, g: Int=>Int)(x: Int) = (f andThen g)(x)
doubleMap: (f: Int => Int, g: Int => Int)(x: Int)Int
scala> doubleMap(_+1, _*2)(3)
res9: Int = 8
```

- If the context isn't clear, you will need parentheses:

```
scala> val ys = xs map _*2
<console>:13: error: missing parameter type for expanded function ((x$1: <error>) =>
xs.map(x$1.$times(2)))
      val ys = xs map _*2
                      ^
scala> val ys = xs map (_*2)
ys: List[Int] = List(2, 4, 6)
```

- If you need to specify the type, you will need braces and, maybe, “case”:

```
scala> val xs: List[Any] = List(1,2,3)
xs: List[Any] = List(1, 2, 3)
scala> val ys: List[Int] = xs map { case x: Int => x * 2 }
ys: List[Int] = List(2, 4, 6)
```


Patterns—Review (1)

- In Scala, pattern-matching plays a big part. Patterns are found:
 - In a variable definition:

```
val Some(x) = Option(methodCall); println(x)
```
 - in a case clause (within a match);

```
case Some(x) => println(x)
```
 - in a lambda;

```
map (x => 2*x)  
map {x: Int => 2*x}
```
 - in a for-comprehension.

```
for (x <- xs) yield x*2  
for (Some(x) <- xos) yield x*2
```

 - *BTW, some of these are very subtly similar (I don't even understand some of the distinctions—I use the source-code analyzer to help in some situations)*
- The important thing is that a pattern not only *matches* but also serves as a pseudo-variable within its scope.

Patterns (2)

- Example:

```
def map[U](f: (T) => U): RandomState[U] =  
  JavaRandomState[U](n, n => f(g(n)))
```

- In this fragment of code, there are two “n”s. The first *n* is a variable in the scope of the *map* method and its enclosing class. The second *n* is a pattern (within a lambda). It could equally have been *x* (probably should have). The lambda could also have been written (with no explicit pattern):

```
f(g(_))
```

- Another form that is basically the same:

```
def map[U](f: (T) => U): RandomState[U] =  
  JavaRandomState[U](n, {m:Long => f(g(m))})
```

Patterns (3) and “_”

- The anonymous match-everything pattern: `_`
 - Similar to a simple identifier like `x`, which also matches everything, the “`_`” does not define a bound variable, e.g. *`case _ => None`*
- But the underscore `_` has several other meanings:
 - Anonymous bound variable in a lambda, e.g. `_+_`
 - The wildcard in an import statement (like “`*`” in Java)
 - Higher-kinded type parameter, e.g. *`def f[M[_]]`*
 - η -expansion of method into function, e.g. *`apply _`*
 - conversion of sequence to varargs: as in *`f(xs: _*)`* or as in *`case Seq(xs @ _*)`*

The IDE is there to help!

- If all of this syntax stuff seems confusing, recall that the IDE and its built-in compiler is there to help you.
- If types are refusing to match, look at some of your intermediate variables and, if necessary, explicitly specify a type annotation (easy to do with an IDE “quick fix,” i.e. the light bulb in IntelliJ IDEA).
- Do you have a strange type being mentioned like *Any* or *Product*? Chances are you are missing the else part of an *if* clause.

Part two

Collections

Let's talk some more about collections

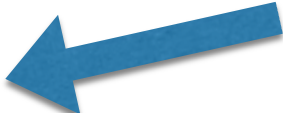
- Real-life software generally involves collections.
 - Sometimes, we want to deal with one thing a time, more usually many things at once.
 - These are the kinds of things we want to do with collections:
 - find if the collection is empty (*isEmpty*);
 - find the number of elements in the collection (*length/size*);
 - (optionally) select a specific element by position or key (*get*);
 - traverse each element in succession, applying a side-effect function to each (*foreach*);
 - create a new collection based on the original, but with only those elements that satisfy a predicate (*filter*);
 - traverse each element in succession, applying a function to each, thus yielding a new collection (*map, flatMap*);
 - traverse each element in succession, applying a function to each element and an accumulator, thus yielding a value (*reduce*);
 - create a new collection based on the original, but with new element(s) (concatenate or “*cons*”).

A simple definition of List

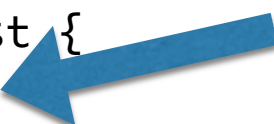
- This is (more or less) what I wrote on the board last time:

```
package edu.neu.coe.scala.list
trait List {
  def length: Int
}
case object Nil extends List {
  def length: Int = 0
}
case class Cons (head: Int, tail: List) extends List {
  def length: Int = 1 + tail.length
}
object List {
  def apply(as: Int*): List =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

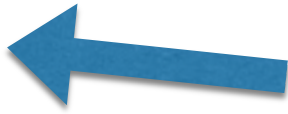
First, we define some behavior in a trait. For now, the only behavior we've defined for our *List* is the ability to get its length.



Remember *proof by induction*? There are two cases: the “base case” and the “inductive step”. We will typically (but not always) have two “case class/object” extensions of a trait.



This is the “companion” object for *List*. Any class or trait can have such a companion object. Case classes always have one (via “syntactic sugar”)



Parametric Types

- A very quick observation before we get into lists.
- We can define a *List* of *Int*, a *List* of *String*, etc.
- But we'd end up having to write all the same methods for each! That would be no good!!
- So, in Scala, all containers have an *underlying* type, including *List*. Such types are known as **Parametric types*** because that's what they are. Scala doesn't really use the term *generics* (partly because it wasn't an afterthought).
- Unlike in Java, we cannot define a *List* without a parametric type because this would break type inference.

* such types make up the *parameters* of a type—and are enclosed in `[]`, just like parameters of a method are enclosed in `()`.

Lists and their methods

- Recap:

```
package edu.neu.coe.scala.list
trait List[+A]
case object Nil extends List[Nothing]
case class Cons[+A] (head: A, tail: List[A]) extends List[A]
object List {
  def apply[A](as: A*): List[A] =
    if (as.isEmpty) Nil
    else Cons(as.head, apply(as.tail: _*))
}
```

We have called the parametric type of the *List* “A”.
A stands for any type, even a *List*[[B]]. I will explain the “+” shortly.

The name for *Cons* in the Scala library is “::”

- What are the methods that we expect *List* to implement?

- Let’s try a few signatures and think what they might mean:

- def x0: Boolean
- def x1: Int
- def x2: A
- def x3: List[A]
- def x4(x: Int): Option[A]
- def x5(f: A=>Boolean): List[A]
- def x6(f: A=>Boolean): Option[A]
- def x7[B](f: A=>B): List[B]
- def x8[B](f: A=>List[B]): List[B]
- def x9(f: A=>Unit): Unit

actually, there are a couple of plausible methods
which yield an A

we are defining an “algebra”
for the *List* type

List methods (“SOE”)

```
def x0: Boolean = this match {case Nil => true; case _ => false }
```


```
def x1: Int = this match {  
  case Nil => 0  
  case Cons(hd, tl) => 1 + tl.x1  
}
```

```
def x2a: A = this match {  
  case Nil => throw new Exception("logic error")  
  case Cons(hd, tl) => hd  
}
```

```
// Alternative interpretation  
def x2b: A = this match {  
  case Nil => unit(0)  
  case Cons(hd, tl) => hd + tl.x2  
}
```

```
def x3: List[A] = this match {  
  case Nil => Nil;  
  case Cons(hd, tl) => tl  
}
```

```
def x4(x: Int): Option[A] = {  
  @tailrec def inner(as: List[A], x: Int): Option[A] = as match {  
    case Nil => None  
    case Cons(hd, tl) => if (x == 0) Some(hd) else inner(tl, x - 1)  
  }  
  if (x < 0) None else inner(this, x)  
}
```



Will not compile: need an operator + that can combine two *A* objects into another *A*. Need *unit* function, too.

List methods (higher-order functions)

- ```
def x5(f: A=>Boolean): List[A] = this match {
 case Cons(hd,tl) => val ftl = tl.x5(f); if (f(hd)) Cons(hd, ftl) else ftl
 case Nil => Nil
}
```
- ```
def x6(f: A=>Boolean): Option[A] = this match {  
  case Cons(hd,tl) => if (f(hd)) Some(hd) else tl.x6(f)  
  case Nil => None  
}
```
- ```
def x7[B](f: A=>B): List[B] = this match {
 case Cons(hd,tl) => Cons(f(hd),tl.x7(f))
 case Nil => List[B]()
}
```
- ```
def ++[B >: A](x: List[B]): List[B] = this match {  
  case Nil => x  
  case Cons(hd,tl) => Cons(hd, tl ++ x)  
}
```
- ```
def x8[B](f: A=>List[B]): List[B] = this match {
 case Cons(hd,tl) => f(hd) ++ tl.x8(f)
 case Nil => List[B]()
}
```
- ```
def x9(f: A=>Unit): Unit = this match {  
  case Cons(hd,tl) => f(hd); tl.x9(f)  
  case Nil => Unit  
}
```

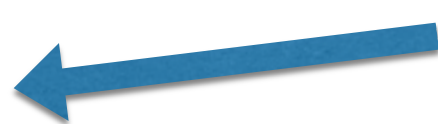
We need a way to concatenate two lists.

But can't we just use *A* as the type of both lists? No: co-/contra-variance

Giving the methods names:

- `def isEmpty: Boolean = this match {case Nil => true; case _ => false }`
- `def length: Int = this match {
 case Nil => 0
 case Cons(hd, tl) => 1 + tl.length
}`
- `def head: A = this match {
 case Nil => throw new Exception("logic error")
 case Cons(hd, tl) => hd
}`
- `def sum: A = this match {
 case Nil => unit(0)
 case Cons(hd, tl) => hd + tl.sum
}`
- `def tail: List[A] = this match {
 case Nil => Nil;
 case Cons(hd, tl) => tl
}`
- `def get(x: Int): Option[A] = {
 @tailrec def inner(as: List[A], x: Int): Option[A] = as match {
 case Nil => None
 case Cons(hd, tl) => if (x == 0) Some(hd) else inner(tl, x - 1)
 }
 if (x < 0) None else inner(this, x)
}`

Will not compile: need an operator + that can combine two *A* objects into another *A*. Need *unit* function, too.



and names for the higher-order functions...

- ```
def filter(f: A=>Boolean): List[A] = this match {
 case Cons(hd,tl) => val ftl = tl.filter(f); if (f(hd)) Cons(hd, ftl) else ftl
 case Nil => Nil
}
```
- ```
def find(f: A=>Boolean): Option[A] = this match {  
  case Cons(hd,tl) => if (f(hd)) Some(hd) else tl.find(f)  
  case Nil => None  
}
```
- ```
def map[B](f: A=>B): List[B] = this match {
 case Cons(hd,tl) => Cons(f(hd),tl.map(f))
 case Nil => List[B]()
}
```
- ```
def ++[B >: A](x: List[B]): List[B] = this match {  
  case Nil => x  
  case Cons(hd,tl) => Cons(hd, tl ++ x)  
}
```
- ```
def flatMap[B](f: A=>List[B]): List[B] = this match {
 case Cons(hd,tl) => f(hd) ++ tl.flatMap(f)
 case Nil => List[B]()
}
```
- ```
def foreach(f: A=>Unit): Unit = this match {  
  case Cons(hd,tl) => f(hd); tl.foreach(f)  
  case Nil => Unit  
}
```

We need a way to concatenate two lists.

But can't we just use *A* as the type of both lists? No: co-/contra-variance

Method categories

- (Refer to my [StackOverflow answer](#) for the original)
- Let's assume a type *Bunch[T]* which extends *Iterable[T]* (the base trait for all Scala collections)
- In the following, *U* is a supertype of *T*, *V* is any *T*-related type:
 - *traversing*: there are actually two subclasses:
 - *shape-preserving*: these define a return type of *Bunch[U]*; example: *map*;
 - *non-shape-preserving*: these define a return type of *Iterator[T]*, *Iterable[T]*, *Iterable[U]*, etc.; example: *iterator*;
 - *side-effecting*: these define a return type of *Unit*; example *foreach*;
 - *selecting*: these define a return type of *T*; example *head*;
 - *maybeSelecting*: these define a return type of *Option[T]*
 - *aggregation*: these define a return type of *V*; example: *foldLeft*;
 - *testing*: these define a return type of *Boolean*; example: *isEmpty*;
 - etc. etc.