Specifications of Scala

Pooja Makula
IS201301014

Scala is a Java-like programming language which unifies object-oriented and functional programming. Scala is a functional language in the sense that every function is a value. Nesting of function definitions and higher-order functions are naturally supported. Scala also supports a general notion of pattern matching which can model the algebraic types used in many functional languages.

Scala has been designed to interoperate seamlessly with Java (an alternative implementation of Scala also works for .NET). Scala classes can call Java methods, create Java objects, inherit from Java classes and implement Java interfaces. None of this requires interface definitions or glue code.

Scala is statically typed

Scala is equipped with an expressive type system that enforces statically that abstractions are used in a safe and coherent manner. In particular, the type system supports:

- generic classes,
- variance annotations,
- upper and lower type bounds,
- inner classes and abstract types as object members,
- compound types,
- explicitly typed self references,
- views, and
- polymorphic methods.

A local type inference mechanism takes care that the user is not required to annotate the program with redundant type information. In combination, these features provide a powerful basis for the safe reuse of programming abstractions and for the type-safe extension of software.

Scala is extensible

In practice, the development of domain-specific applications often requires domain-specific language extensions. Scala provides a unique combination of language mechanisms that make it easy to smoothly add new language constructs in form of libraries:

- any method may be used as an infix or postfix operator, and
- closures are constructed automatically depending on the expected type (target typing).

A joint use of both features facilitates the definition of new statements without extending the syntax and without using macro-like meta-programming facilities.

Lexical Syntax: Scala programs are written using the Unicode Basic Multilingual Plane (BMP) character set. There are two modes of Scala's lexical syntax, the Scala mode and the XML mode.

(i) Multiline String Literals: A multi-line string literal is a sequence of characters enclosed in triple quotes """ ... """. The sequence of characters is arbitrary, except that it may contain three or more consecutive quote characters only at the very end.
Syntax: stringLiteral ::= '"""' multiLineChars '"""' multiLineChars ::= {['"'] ['"'] charNoDoubleQuote} {'"'}
The Scala library contains a utility method stripMargin which can be used to strip leading whitespace from multi-line strings.
The expression """the present string |spans three |lines.""".stripMargin
evaluates to
the present string
spans three
lines.
(ii) XML mode: In order to allow literal inclusion of XML fragments, lexical analysis switches from Scala mode to XML mode when encountering an opening angle bracket '<' in the following circumstance: The '<' must be preceded either by whitespace, an opening parenthesis or an opening brace and immediately followed by a character starting an XML name.
Syntax: (whitespace | '(' | '{') '<' (XNameStart | '!' | '?')
XNameStart ::= '_' | BaseChar | Ideographic *(asinW3CXML,butwithout* ':'

Scala Properties:

1) **Generic Classes:** Scala has built-in support for classes parameterized with types. Such generic classes are particularly useful for the development of collection classes. Class Stack models imperative (mutable) stacks of an arbitrary element type T. The use of type parameters allows to check that only legal elements (that are of type T) are pushed onto the stack. Similarly, with type parameters we can express that method top will only yield elements of the given type.

   Subtyping of generic types is *invariant*. This means that if we have a stack of characters of typeStack[Char] then it cannot be used as an integer stack of type Stack[Int]. This would be unsound because it would enable us to enter true integers into the character stack. To conclude, *Stack[T]* is only a subtype of *Stack[S]* iff *S = T*. Since this can be quite restrictive, Scala offers a type parameter annotation mechanism to control the subtyping behavior of generic types.

2) Variance Annotations: Scala supports variance annotations of type parameters of generic classes. In contrast to Java 5 (aka. JDK 1.5), variance annotations may be added when a class abstraction is *defined*, whereas in Java 5, variance annotations are given by clients when a class abstraction is *used*. There can be a functional (i.e. immutable) implementation for stacks which does not have this restriction.

3) Upper and Lower type bounds: In Scala, type parameters and abstract types may be constrained by a *type bound*. Such type bounds limit the concrete values of the type variables and possibly reveal more information about the members of such types. An *upper type bound* T <: Adeclares that type variable T refers to a subtype of type A.

While upper type bounds limit a type to a subtype of another type, lower type bounds declare a type to be a supertype of another type. The term T >: A expresses that the type parameter T or the abstract type T refer to a supertype of type A.

4) Inner classes and abstract types as object members: In Scala it is possible to let classes have other classes as members. Opposed to Java-like languages where such inner classes are members of the *enclosing class*, in Scala such inner classes are bound to the *outer object*.

In Scala, classes are *parameterized* with values (the constructor parameters) and with types (if classes are generic). For reasons of regularity, it is not only possible to have values as *object members*; types along with values are members of objects. Furthermore, both forms of members can be concrete and abstract.

Here is an example which defines both a deferred value definition and an *abstract type* definition as members of class Buffer.

```
abstract class Buffer {
  type T
  val element: T
}
```

Abstract types are types whose identity is not precisely known. In the example above, we only know that each object of class Buffer has a type member T, but the definition of class Buffer does not reveal to what concrete type the member type T corresponds. Like value definitions, we can override type definitions in subclasses. This allows us to reveal more information about an abstract type by tightening the type bound (which describes possible concrete instantiations of the abstract type).

In the following program we derive a class SeqBuffer which allows us to store only sequences in the buffer by stating that type T has to be a subtype of Seq[U] for a new abstract type U:

```
abstract class SeqBuffer extends Buffer {
  type U
  type T <: Seq[U]
  def length = element.length
}
```

Traits or classes with abstract type members are often used in combination with anonymous class instantiations.

5) Compound Types: Sometimes it is necessary to express that the type of an object is a subtype of several other types. In Scala this can be expressed with the help of *compound types*, which are intersections of object types.

Suppose we have two traits Cloneable and Resetable:

```
trait Cloneable extends java.lang.Cloneable {
  override def clone(): Cloneable = { super.clone(); this }
}
trait Resetable {
  def reset: Unit
}
```

Compound types can consist of several object types and they may have a single *refinement* which can be used to narrow the signature of existing object members.

The general form is: *A* **with** *B* **with** *C* ... { *refinement* }

```
def cloneAndReset(obj: Cloneable with Resetable): Cloneable = {
  //...
}
```

6) Explicitly typed self references: In Scala it is possible to tie a class to another type (which will be implemented in future) by giving self reference this the other type explicitly. The explicit self type is specified within the body of the class DirectedGraph**.**

```
abstract class DirectedGraph extends Graph {
  ...
  class NodeImpl extends NodeIntf {
    self: Node =>
    def connectWith(node: Node): Edge = {
      val edge = newEdge(this, node)  // now legal
      edges = edge :: edges
      edge
    }
  }
  ...
}
```

7) Views: A view from type S to type T is defined by an implicit value which has function type S => T, or by a method convertible to a value of that type.

Views are applied in two situations:

- If an expression e is of type T, and T does not conform to the expression's expected type pt.
- In a selection e.m with e of type T, if the selector m does not denote a member of T.

In the first case, a view v is searched which is applicable to e and whose result type conforms to pt. In the second case, a view v is searched which is applicable to e and whose result contains a member named m.

The following operation on the two lists xs and ys of type List[Int] is legal:

```
xs <= ys
```

assuming the implicit methods list2ordered and int2ordered defined below are in scope:

```
implicit def list2ordered[A](x: List[A])
    (implicit elem2ordered: a => Ordered[A]): Ordered[List[A]] =
  new Ordered[List[A]] { /* .. */ }

implicit def int2ordered(x: Int): Ordered[Int] =
  new Ordered[Int] { /* .. */ }
```

The list2ordered function can also be expressed with the use of a *view bound* for a type parameter:

```
implicit def list2ordered[A <% Ordered[A]](x: List[A]): Ordered[List[A]] = ...
```

8) Polymorphic Methods: Methods in Scala can be parameterized with both values and types. Like on the class level, value parameters are enclosed in a pair of parentheses, while type parameters are declared within a pair of brackets.

Here is an example:

```
object PolyTest extends Application {
  def dup[T](x: T, n: Int): List[T] =
```

```
    if (n == 0) Nil
    else x :: dup(x, n - 1)
  println(dup[Int](3, 4))
  println(dup("three", 3))
}
```

Method dup in object PolyTest is parameterized with type T and with the value parameters x: T and n: Int. When method dup is called, the programmer provides the required parameters (see line 5 in the program above), but as line 6 in the program above shows, the programmer is not required to give actual type parameters explicitly. The type system of Scala can infer such types. This is done by looking at the types of the given value parameters and at the context where the method is called.

Please note that the trait Application is designed for writing short test programs, but should be avoided for production code as it may affect the ability of the JVM to optimize the resulting code; please use def main() instead.

9) Infix or postfix operator: Any method which takes a single parameter can be used as an infix operator in Scala. Here is the definition of classMyBool which defines three methods and, or, and negate.

```
class MyBool(x: Boolean) {
  def and(that: MyBool): MyBool = if (x) that else this
  def or(that: MyBool): MyBool = if (x) this else that
  def negate: MyBool = new MyBool(!x)
}
```

It is now possible to use and and or as infix operators:

```
def not(x: MyBool) = x negate; // semicolon required here
def xor(x: MyBool, y: MyBool) = (x or y) and not(x and y)
```

10) Automatic Type-Dependent Closure Construction: Scala allows parameterless function names as parameters of methods. When such a method is called, the actual parameters for parameterless function names are not evaluated and a nullary function is passed instead which encapsulates the computation of the corresponding parameter (so-called *call-by-name* evalutation).

The following code demonstrates this mechanism:

```
object TargetTest1 extends Application {
  def whileLoop(cond: => Boolean)(body: => Unit): Unit =
    if (cond) {
      body
      whileLoop(cond)(body)
    }
  var i = 10
  whileLoop (i > 0) {
    println(i)
    i -= 1
  }
}
```

The function whileLoop takes two parameters cond and body. When the function is applied, the actual parameters do not get evaluated. But whenever the formal parameters are used in the body of whileLoop, the implicitly created nullary functions will be evaluated instead. Thus, our method whileLoop implements a Java-like while-loop with a recursive implementation scheme.