Scala−−
Language Reference Manual

Contents

1 Introdution

This manual describes Scala−−, a primitive static scoping, strong static typing, type inferred, functional programming language with immutable data structures, simple pattern matching and specialized emphasis on type inference. Scala−− written in OCaml syntactically resembles a small subset of Scala functionalities with educational purpose, that compiles to the LLVM Intermediate Representation. Several standard library functions and data structures including `map`, `filter`, List and Map are implemented in the Scala−−, providing relatively advanced immutabe operations, demonstrated the possibility of becoming a general purpose programming language. This manual describes in detail the lexical conventions, type systems, scoping rules, standard library functions, data structures and the gramar of the Scala−− language.

2 Lexical syntax

2.1 Identifiers

In Scala−−, there is only one way to form an identifier. It can start with a letter which can be followed by an arbitrary sequence of letters and digits, which may be followed by underscore '_' characters and following similar sequences, which can be defined by the following regular expression:

```
['a' - 'z' 'A' - 'Z' ]['a' - 'z' 'A' - 'Z' '0' - '9' '_' ]*
```

2.2 Keywords

Scala−− has a set of reserved keywords that can not be used as identifiers.

2.2.1 Statements and blocks

The following keywords indicate types of statement or blocks:

```
var val object final
```

2.2.2 Functions

The following keyword is reserved for function-related purpose:

```
def main
```

2.2.3 Pattern matching

The following keywords are reserved for pattern matching-related purposes:

```
match with case Some _
```

2.2.4 Control flow

The following keywords are used for control flow:

```
if else return for do while yield until to break <-
```

2.2.5 Types

The following primitive type keywords resemble Scala's object type:

```
Int Float Char Boolean :
```

2.2.6 Built-in functions

The following keywords are reserved for built-in functions and/or data structures:

```
random, max, min, floor, map, filter, println, Map, List,
Array, Tuple, insert, remove, ++, ++=, ->
```

2.3 Literals

Scala−− supports boolean, integer, float, character, string, escape sequence, symbol literals.

2.3.1 Integer

The following regular expression defines a decimal digit:

```
digit = ['0' - '9']
```

An integer of type Int is a 64-bit signed immutable value consisting of at least one digit taking the following form:

```
digit+
```

2.3.2 Floating point

Floating point numbers can be represented as exponentials as the following regular exressions:

```
exp = 'e' ['+' '-']?  ['0' - '9']+
```

So in general, floating point numbers take the following regular expression:

```
digit'.'digit[exp] | '.'digit[exp] | digit exp | digit [exp]
```

2.3.3 Boolean

Boolean literals are the following:

```
boolean = ["true" | "false"]
```

2.3.4 Character

Characters are single, 8-bit, ASCII symbols.

```
char = ['a' - 'z'] | ['A' - 'Z']
```

2.3.5 String

A string literal is a sequence of characters in double quotes.

2.3.6 Escape sequence

The following escape sequences are recognized in character and string literals:

\b	backspace
\t	horizontal tab
\n	linefeed
\r	carriage return
\"	double quote
\'	single quote
\\	backslash

2.4 Punctions

Punctions group and/or separate the primary expressions consisting of above-mentioned identifiers and literals.

`()` can indicates a list of arguments in a function declaration or function call; it can also be position access operator in built-in facilities of Array, Map and List; it can also be the boundry symbols of built-in facilities of Array, Map, List and Tuple.

`{}` defines statement blocks.

`,` represents the separator between a list of arguments in functions or a list of items in built-in data structures.

`;` is a newline character separating expressions and statements.

2.5 Comments and whitespace

Comments in Scala−− start with /* and terminate with */, where multiple line comments are not allowed to be nested. Single line comments take the form of //.

2.6 Operations

Scala−− supports several operations including artithmetic and booleans literals.

2.6.1 Value binding

A single equsl sign indicates assignment operation or in an assignment or declaration statement:

```
=
```

2.6.2 Operators

The following binary operators are supported in Scala−−:

```
/, *, %
==, !=, <=, <, >, >=
&, |
```

The following unary operators are supported:

```
!, ~
```

The following operators can be either bintary or unary, depening the context:

```
+, -
```

3 Syntax

3.1 Program structure

Scala−− program consists of decarations which are made of optional global variable and/or newline-separated and/or optionally semi-colon-separated function decarations as well as function bodies which may include variable assignment or nested function bodies.

```
program:
   declaration
   program declaration
declaration:
   fundec
   vardec newline
   newline
```

3.1.1 Variable declarations and type inference

Variables can be declared and initialized globally, or locally in a function body:

`<var|val> <id-list> [ : var-type ] [ = value ] [;] <nl>`, where "`nl`" represents newline.

Type inference Instead of *advanced local type inference* algorithm employed in Scala, Scala−− experimented type inference using one type of *complete type inference* algorithm – *Hindley-Milner type inference* algorithm which has been broadly adopted in a spate of contemporary type inferred functional programming languages such as Standard ML, OCaml and Haskell.

A variable can be declaraed as the following:

```
val myInt : Int = 17
val myFloat : Float = 3.14
val myChar : Char = 'c'
val myString : String = "Hello World!"
val myBoolean : Boolean = true
val myList : List[Int] = List(1,1,2,3,5,8)
val myMap : Map = Map( "Static typing" -> "OCaml", "Dynamic
    typing" -> "Elixir")
```

The above expressions are equvivlent to the following:

```
val myInt = 17
val myFloat = 3.14
val myChar = 'c'
val myString = "Hello World!"
val myBoolean = true
val myList = List(1,1,2,3,5,8)
val myMap = Map( "Static typing" -> "OCaml", "Dynamic typing"
    -> "Elixir")
```

Correctness of type inference also hold true when apply to function declarations regarding formal arguments and function return type which

is documented in following sections.

Mutable and immutable variables Immutable variables are defined with keyword `val`, while mutable variables are defined with keyword `var` as the following:

```
val immutList = List("I" "Can" "NOT" "Be" "Modified" "!")
var mutString = "I am ok to be changed."
```

3.1.2 Function declarations

Functions are defined in the following way:

```
def func-id (formal-listopt) [ :  var-typeopt ] block
```

Here `def` is a keyword starting a function declaration or definition. func-id is the identifier of a instance of the function. `block` contains the function body. `formal-listopt` is optionally required when decaring or defining a function, containing the formal arguments of `var-type`. HM type inference algorithm applies here.

```
formal-list:
   formal-type-cluster
   formal-list, formal-type-cluster
formal-type-cluster:
   var-type
   id-list var-type
```

For instance, a function can be declaraed with explicit specification of the argument types:

```
/* Return summation of two integer numbers */
def sumOfSquares(x: Int, y: Int): Int = {
    val x2 = x * x
        val y2 = y * y
        x2 + y2
}
```

3.2 Expressions

3.2.1 Primary expressions

Primary expressions consist of literals and parathesized expression.

3.2.2 Precedence of operations

3.3 Statements

3.3.1 Assignments

Assignment of variables requires using = keyword. For example:

```
val anInt = 5
var aChar = 'w'
```

3.3.2 Blocks and control flow

Conditional in Scala−−:

```
// If statements are like Java except they return a value
    like the ternary operator
// Conditional operators: ==, !=, >, <, <=, >=
// Logical operators: &&, ||, !

val age = 18
val canVote = if (age >= 18) "yes" else "no"

// {} is required in the REPL, but not otherwise
if ((age >= 5) && (age <= 6)) {
   println("Go to Kindergarten")
} else if ((age > 6) && (age <= 7)) {
   println("Go to Grade 1")
} else {
   println("Go to Grade " + (age - 5))
}

true || false
!(true)
```

There are for-loop, while-loop, if-else statement in Scala−−.

```
/* Scala-- while-loop example: */
   var i = 0;
   while (i <= 5) {
      println(i)
      i += 1
   }

/* Scala-- do-while-loop example: */
   do {
      println(i)
      i += 1
   } while (i <= 9)

/* Scala-- for-loop example: */
   for (i <- 1 to 10) {
      println(i)
   }

// until is often used to loop through strings or arrays
      val randLetters = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
      //for (i <- 0 to (randLetters.length - 1)) {
      for (i <- 0 until randLetters.length) {
// Get the letter in the index of the String
         println(randLetters(i))
      }

// Used to iterate through a list
      val aList = List(1,2,3,4,5)
      for (i <- aList) {
         println("List iitem " + i)
      }
```

```
// Store even numbers in a list
     var evenList = for {
        i <- 1 to 20
// You can put as many conditions here separated with
    semicolons as you need
        if (i % 2) == 0
        if (i % 3) == 0
     } yield i

     println("Even list:")
     for (i <- evenList)
        println(i)

// This loop assigns a value to the 1st variable and it
    retains that value until the 2nd finishes its cycle and
    then it iterates
     for (i <- 1 to 5; j <- 6 to 10) {
        println("i: " + i)
        println("j: " + j)
     }
```

4 Scoping rules

Scala−− uses lexical scoping just as Scala does. The scope of an variable or a function is limited to the block where it is declared, if it is not a global variable or function.

5 Standard library and collections

Several standard library functions and data structures are implemented in Scala−− to provide feature-rich coding experience.

5.1 println, map and filter

The following code shows an example using map and filter:

```
    // "_" is used to indicate the meant function when
        declared in order for subsequent passing function as
        an argument
    val log10Func = log10 _
    println("Log10 is: " + log10Func(1000))
    // Apply a function to all items of a list with map
    List(1000.0, 10000.0).map(log10Func).foreach(println)

    // Filter passes only those values that meet a condition
    List(1,2,3,4).filter(_ % 2 == 0).foreach(println)
```

5.2 Array

Built-in Array has several operations to ease the manipulation, that includes `insert`, `remove`, `empty`, `+=`, `++=`.

```
    // Create and initialize array
    val friends = Array("Bob", "Tom")
    // Change the value in an array
    friends(0) = "Sue"
    println("Best friends: " + friends(0))
    // Create an ArrayBuffer
    val friends2 = ArrayBuffer[String]()
    // Add an item to the 1st index
    friends2.insert(0, "Phil")
    // Add item to the next available slot
    friends2 += "Mark"
    // Add multiple values to the next available slot
    friends2 ++= Array("Susy", "Paul")
    // Add items starting at 2nd slot
    friends2.insert(1, "Mike", "Sally", "Sam")
    // Remove the 2nd element
    friends2.remove(1)
    // Remove two elements starting at the 2nd index
    friends2.remove(1, 2)
```

5.3 List

List examplifies as the following:

```
    val aList = List(1,2,3,4,5)
```

6 Code example

6.1 Hello World

```
/* Hello World Example in Scala-- */
/**
 * Author: _________________
 * Description: ____________
 * Last modified:___________
 * Usage: __________________
 */
Object HelloWorld {
   // Function entry point starts here:
   def main(args: Array[String])
   /
      pritnln("Hello, world!")
   }
}
```

7 Reference

1. V. Cremet, F. Garillot, S. Lenglet, and M. Odersky. A core calculus for scala type checking. *Mathematical Foundations of Computer Science*, 1-23, 2006.

2. M. Odersky, The Scala Language Spcification, *Programming Methods Laboratory*, EPFL, Switzerland, 2014.

3. N. AlDuaij, S. N. Farra, Y. Kang, A. Lottarini, Funk Programming Language Reference Manual, *Course of W4115: Programming language translators*, Columbia University, 2012.