Click here to download PDF version of specification

Table of Contents

Authors and Contributors

Martin Odersky, Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Philipp Haller, Stéphane Micheloud, Nikolay Mihaylov, Adriaan Moors, Lukas Rytz, Michel Schinz, Erik Stenman, Matthias Zenger

Markdown Conversion by Iain McGinniss.

Preface

Scala is a Java-like programming language which unifies object-oriented and functional programming. It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Classes can be composed using mixin composition.
Scala is designed to work seamlessly with less pure but mainstream object-oriented languages like Java.

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. 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 has been developed from 2001 in the programming methods laboratory at EPFL. Version 1.0 was released in November 2003. This document describes the second version of the language, which was released in March 2006. It acts a reference for the language definition and some core library modules. It is not intended to teach Scala or its concepts; for this there are other documents.

Scala has been a collective effort of many people. The design and the implementation of version 1.0 was completed by Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and the author. Iulian Dragos, Gilles Dubochet, Philipp Haller, Sean McDirmid, Lex Spoon, and Geoffrey Washburn joined in the effort to develop the second version of the language and tools. Gilad Bracha, Craig Chambers, Erik Ernst, Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Meijer, Klaus Ostermann, Didier Rémy, Mads Torgersen, and Philip Wadler have shaped the design of the language through lively and inspiring discussions and comments on previous versions of this document. The contributors to the Scala mailing list have also given very useful feedback that helped us improve the language and its tools.

Lexical Syntax

Scala programs are written using the Unicode Basic Multilingual Plane (*BMP*) character set; Unicode supplementary characters are not presently supported.
This chapter defines the two modes of Scala's lexical syntax, the Scala mode and the *XML mode*.
If not otherwise mentioned, the following descriptions of Scala tokens refer to *Scala mode*, and literal characters (marked as `c`) refer to the ASCII fragment `\u0000 – \u007F`.

In Scala mode, *Unicode escapes* are replaced by the corresponding Unicode character with the given hexadecimal code.

```
UnicodeEscape ::= ‘\’ ‘u’ {‘u’} hexDigit hexDigit hexDigit hexDigit
hexDigit      ::= ‘0’ | … | ‘9’ | ‘A’ | … | ‘F’ | ‘a’ | … | ‘f’
```

To construct tokens, characters are distinguished according to the following classes (Unicode general category given in parentheses):

1. Whitespace characters. `\u0020 | \u0009 | \u000D | \u000A`.
2. Letters, which include lower case letters (`Ll`), upper case letters (`Lu`), titlecase letters (`Lt`), other letters (`Lo`), letter numerals (`Nl`) and the two characters `\u0024 "$"` and `\u005F "_"`.
3. Digits `"0" | … | "9"`.
4. Parentheses `"(" | ")" | "[" | "]" | "{" | "}"`.
5. Delimiter characters `"`" | "'" | """ | "." | ";" | ","`.
6. Operator characters. These consist of all printable ASCII characters (`\u0020 - \u007E`) that are in none of the sets above, mathematical symbols (`Sm`) and other symbols (`So`).

Identifiers

```
op       ::=  opchar {opchar}
varid    ::=  lower idrest
boundvarid ::=  varid
             | ‘`’ varid ‘`’
plainid  ::=  upper idrest
           |  varid
           |  op
id       ::=  plainid
           |  ‘`’ { charNoBackQuoteOrNewline | UnicodeEscape | charEscapeS
idrest   ::=  {letter | digit} [‘_’ op]
```

There are three ways to form an identifier. First, an identifier can start with a letter which can be followed by an arbitrary sequence of letters and digits. This may be followed by underscore `'_'` characters and another string composed of either letters and digits or of operator characters. Second, an identifier can start with an operator character followed by an arbitrary sequence of operator characters. The preceding two forms are called *plain* identifiers. Finally, an identifier may also be formed by an arbitrary string between back-quotes (host systems may impose some restrictions on which strings are legal for identifiers). The identifier then is composed of all characters excluding the backquotes themselves.

As usual, a longest match rule applies. For instance, the string `big_bob++=`def`` decomposes into the three identifiers `big_bob`, `++=`, and `def`.

Identifiers, Names and Scopes

Names in Scala identify types, values, methods, and classes which are collectively called *entities*. Names are introduced by local definitions and declarations, inheritance, import clauses, or package clauses which are collectively called *bindings*.

Bindings of different kinds have a precedence defined on them:

1. Definitions and declarations that are local, inherited, or made available by a package clause and also defined in the same compilation unit as the reference to them, have highest precedence.
2. Explicit imports have next highest precedence.
3. Wildcard imports have next highest precedence.
4. Definitions made available by a package clause, but not also defined in the same compilation unit as the reference to them, as well as imports which are supplied by the compiler but not explicitly written in source code, have lowest precedence.

There are two different name spaces, one for types and one for terms. The same name may designate a type and a term, depending on the context where the name is used.

A binding has a *scope* in which the entity defined by a single name can be accessed using a simple name. Scopes are nested. A binding in some inner scope *shadows* bindings of lower precedence in the same scope as well as bindings of the same or lower precedence in outer scopes.

Note that shadowing is only a partial order. In the following example, neither binding of `x` shadows the other. Consequently, the reference to `x` in the last line of the block is ambiguous.

```
val x = 1
locally {
  import p.X.x
  x
}
```

Function Types

```
Type              ::=  FunctionArgs ‘=>’ Type
FunctionArgs      ::=  InfixType
                     |  ‘(’ [ ParamType {‘,’ ParamType } ] ‘)’
```

The type $(T_1, \ldots, T_n) \Rightarrow U$ represents the set of function values that take arguments of types $T1, \ldots, Tn$ and yield results of type U. In the case of exactly one argument type $T \Rightarrow U$ is a shorthand for $(T) \Rightarrow U$. An argument type of the form $\Rightarrow T$ represents a call-by-name parameter of type T.

Function types associate to the right, e.g. $S \Rightarrow T \Rightarrow U$ is the same as $S \Rightarrow (T \Rightarrow U)$.

Function types are shorthands for class types that define `apply` functions. Specifically, the n-ary function type $(T_1, \ldots, T_n) \Rightarrow U$ is a shorthand for the class type `Function`$_n$`[`$-T_1$`, ..., `T_n`, U]`. Such class types are defined in the Scala library for n between 0 and 22 as follows.

```
package scala
trait Functionn[-T1, ..., -Tn, +R] {
  def apply(x1: T1, ..., xn: Tn): R
  override def toString = "<function>"
}
```

Basic Declarations and Definitions

TODO

Classes and Objects

TODO

Expressions

TODO

Implicits

TODO

Pattern Matching

TODO

Top-Level Definitions

TODO

XML Expressions and Patterns

TODO

Annotations

TODO

The Scala Standard Library

TODO

Syntax Summary

TODO

References

TODO

Changelog

TODO