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 | charEsca
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`.

The rules for pattern matching further distinguish between *variable identifiers*, which start with a lower case letter, and *constant identifiers*, which do not. For this purpose, underscore `'_'` is taken as lower case, and the '$' character is taken as upper case.

The '$' character is reserved for compiler-synthesized identifiers. User programs should not define identifiers which contain '$' characters.

The following names are reserved words instead of being members of the syntactic class `id` of lexical identifiers.

```
abstract    case        catch       class       def
do          else        extends     false       final
finally     for         forSome     if          implicit
import      lazy        macro       match       new
null        object      override    package     private
protected   return      sealed      super       this
throw       trait       try         true        type
val         var         while       with        yield
_    :    =    =>    <-    <:    <%    >:    #    @
```

The Unicode operators `\u21D2` ⇒ and `\u2190` ←, which have the ASCII equivalents `=>` and `<-`, are also reserved.

Example
Here are examples of identifiers:

```
x         Object        maxIndex   p2p      empty_?
+         `yield`       #####     _y       dot_product_*
__system  _MAX_LEN_
```

Newline Characters

```
semi ::= ‘;’ |  nl {nl}
```

Scala is a line-oriented language where statements may be terminated by semi-colons or newlines. A newline in a Scala source text is treated as the special token ‘nl’ if the three following criteria are satisfied:

1. The token immediately preceding the newline can terminate a statement.
2. The token immediately following the newline can begin a statement.
3. The token appears in a region where newlines are enabled.

The tokens that can terminate a statement are: literals, identifiers and the following delimiters and reserved words:

```
this    null    true    false    return    type    <xml-start>
_       )       ]       }
```

The tokens that can begin a statement are all Scala tokens *except* the following delimiters and reserved words:

```
catch    else    extends    finally    forSome    match
with    yield    ,    .    ;    :    =    =>    <-    <:    <%
>:    #    [    )    ]    }
```

A `case` token can begin a statement only if followed by a `class` or `object` token.

Newlines are enabled in:

1. all of a Scala source file, except for nested regions where newlines are disabled
2. the interval between matching `{` and `}` brace tokens, except for nested regions where newlines are disabled

Newlines are disabled in:

1. the interval between matching `(` and `)` parenthesis tokens, except for nested regions where newlines are enabled
2. the interval between matching `[` and `]` bracket tokens, except for nested regions where newlines are enabled
3. the interval between a `case` token and its matching `=>` token, except for nested regions where newlines are enabled
4. any regions analyzed in XML mode

Note that the brace characters of `{...}` escapes in XML and string literals are not tokens, and therefore do not enclose a region where newlines are enabled.

Normally, only a single `nl` token is inserted between two consecutive non-newline tokens which are on different lines, even if there are multiple lines between the two tokens. However, if two tokens are separated by at least one completely blank line (i.e a line which contains no printable characters), then two `nl` tokens are inserted.

The Scala grammar (given in full here) contains productions where optional `nl` tokens, but not semicolons, are accepted. This has the effect that a newline in one of these positions does not terminate an expression or statement. These positions can be summarized as follows:

Multiple newline tokens are accepted in the following places (note that a semicolon in place of the newline would be illegal in every one of these cases):

- between the condition of a conditional expression or while loop and the next following expression
- between the enumerators of a for-comprehension and the next following expression
- after the initial `type` keyword in a type definition or declaration

A single new line token is accepted

- in front of an opening brace '{', if that brace is a legal continuation of the current statement or expression
- after an infix operator, if the first token on the next line can start an expression
- in front of a parameter clause
- after an annotation

Example

The newline tokens between the two lines are not treated as statement separators.

```
if (x > 0)
  x = x - 1

while (x > 0)
  x = x / 2

for (x <- 1 to 10)
  println(x)

type
  IntList = List[Int]
```

Example

```
new Iterator[Int]
{
  private var x = 0
  def hasNext = true
  def next = { x += 1; x }
}
```

With an additional newline character, the same code is interpreted as an object creation followed by a local block:

```
new Iterator[Int]

{
  private var x = 0
  def hasNext = true
  def next = { x += 1; x }
}
```

Example

```
x < 0 ||
x > 10
```

With an additional newline character, the same code is interpreted as two expressions:

```
x < 0 ||

x > 10
```

Example

```
def func(x: Int)
        (y: Int) = x + y
```

With an additional newline character, the same code is interpreted as an abstract function definition and a syntactically illegal statement:

```
def func(x: Int)

        (y: Int) = x + y
```

Example

```
@serializable
protected class Data { ... }
```

With an additional newline character, the same code is interpreted as an attribute and a separate statement (which is syntactically illegal).

```
@serializable

protected class Data { ... }
```

Literals

There are literals for integer numbers, floating point numbers, characters, booleans, symbols, strings. The syntax of these literals is in each case same as in Java.

```
Literal  ::=  ['-'] integerLiteral
           |  ['-'] floatingPointLiteral
           |  booleanLiteral
           |  characterLiteral
           |  stringLiteral
           |  interpolatedString
           |  symbolLiteral
           |  'null'
```

Integer Literals

```
integerLiteral  ::=  (decimalNumeral | hexNumeral)
                      [‘L’ | ‘l’]
decimalNumeral  ::=  ‘0’ | nonZeroDigit {digit}
hexNumeral      ::=  ‘0’ (‘x’ | ‘X’) hexDigit {hexDigit}
digit           ::=  ‘0’ | nonZeroDigit
nonZeroDigit    ::=  ‘1’ | … | ‘9’
```

Integer literals are usually of type `Int`, or of type `Long` when followed by a `L` or `l` suffix. Values of type `Int` are all integer numbers between -2^{31} and $2^{31}-1$, inclusive. Values of type `Long` are all integer numbers between -2^{63} and $2^{63}-1$, inclusive. A compile-time error occurs if an integer literal denotes a number outside these ranges.

However, if the expected type *pt* of a literal in an expression is either `Byte`, `Short`, or `Char` and the integer number fits in the numeric range defined by the type, then the number is converted to type *pt* and the literal's type is *pt*. The numeric ranges given by these types are:

`Byte`	-2^7 to 2^7-1
`Short`	-2^{15} to $2^{15}-1$
`Char`	0 to $2^{16}-1$

Example

```
0          21          0xFFFFFFFF       -42L
```

Floating Point Literals

```
floatingPointLiteral  ::=  digit {digit} ‘.’ digit {digit} [exponentPart]
                        |  ‘.’ digit {digit} [exponentPart] [floatType]
                        |  digit {digit} exponentPart [floatType]
                        |  digit {digit} [exponentPart] floatType
exponentPart          ::=  (‘E’ | ‘e’) [‘+’ | ‘-’] digit {digit}
floatType             ::=  ‘F’ | ‘f’ | ‘D’ | ‘d’
```

Floating point literals are of type `Float` when followed by a floating point type suffix `F` or `f`, and are of type `Double` otherwise. The type `Float` consists of all IEEE 754 32-bit single-precision binary floating point values, whereas the type `Double` consists of all IEEE 754 64-bit double-precision binary floating point values.

If a floating point literal in a program is followed by a token starting with a letter, there must be at least one intervening whitespace character between the two tokens.

Example

```
0.0        1e30f      3.14159f      1.0e-100      .1
```

Example

The phrase `1.toString` parses as three different tokens: the integer literal `1`, a `.`, and the identifier `toString`.

Example

`1.` is not a valid floating point literal because the mandatory digit after the `.` is missing.

Boolean Literals

```
booleanLiteral  ::=  ‘true’ | ‘false’
```

The boolean literals `true` and `false` are members of type `Boolean`.

Character Literals

```
characterLiteral  ::=  ‘’’ (charNoQuoteOrNewline | UnicodeEscape | charEsc
```

A character literal is a single character enclosed in quotes. The character can be any Unicode character except the single quote delimiter or `\u000A` (LF) or `\u000D` (CR); or any Unicode character represented by either a Unicode escape or by an escape sequence.

Example

```
'a'    '\u0041'    '\n'    '\t'
```

Note that although Unicode conversion is done early during parsing, so that Unicode characters are generally equivalent to their escaped expansion in the source text, literal parsing accepts arbitrary Unicode escapes, including the character literal `'\u000A'`, which can also be written using the escape sequence `'\n'`.

String Literals

```
stringLiteral  ::=  ‘"’ {stringElement} ‘"’
stringElement  ::=  charNoDoubleQuoteOrNewline | UnicodeEscape | charEscap
```

A string literal is a sequence of characters in double quotes. The characters can be any Unicode character except the double quote delimiter or `\u000A` (LF) or `\u000D` (CR); or any Unicode character represented by either a Unicode escape or by an escape sequence.

If the string literal contains a double quote character, it must be escaped using `"\""`.

The value of a string literal is an instance of class `String`.

Example

```
"Hello, world!\n"
"\"Hello,\" replied the world."
```

Multi-Line String Literals

```
stringLiteral   ::=  ‘"""’ multiLineChars ‘"""’
multiLineChars  ::=  {[‘"’] [‘"’] charNoDoubleQuote} {‘"’}
```

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. Characters must not necessarily be printable; newlines or other control characters are also permitted. Unicode escapes work as everywhere else, but none of the escape sequences here are interpreted.

Example

```
"""the present string
   spans three
   lines."""
```

This would produce the string:

```
the present string
     spans three
     lines.
```

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.
```

Method `stripMargin` is defined in class scala.collection.immutable.StringLike. Because there is a predefined implicit conversion from `String` to `StringLike`, the method is applicable to all strings.

Interpolated string

```
interpolatedString  ::= alphaid ‘"’ {printableChar (‘"’ | ‘$’) | escape} ‘
                      |  alphaid ‘"""’ {[‘"’] [‘"’] char (‘"’ | ‘$’) | esc
escape              ::= ‘$$’
                      | ‘$’ id
                      | ‘$’ BlockExpr
alphaid             ::= upper idrest
                      |  varid
```

Interpolated string consists of an identifier starting with a letter immediately followed by a string literal. There may be no whitespace characters or comments between the leading identifier and the opening quote ‘"’ of the string. The string literal in a interpolated string can be standard (single quote) or multi-line (triple quote).

Inside an interpolated string none of the usual escape characters are interpreted (except for unicode escapes) no matter whether the string literal is normal (enclosed in single quotes) or multi-line (enclosed in triple quotes). Instead, there are two new forms of dollar sign escape. The most general form encloses an expression in ‘${’ and ‘}’, i.e. `${expr}` . The expression enclosed in the braces that follow the leading ‘$’ character is of syntactical category `BlockExpr` . Hence, it can contain multiple statements, and newlines are significant. Single ‘$’-signs are not permitted in isolation in a interpolated string. A single ‘$’-sign can still be obtained by doubling the ‘$’ character: ‘$$’.

The simpler form consists of a ‘$’-sign followed by an identifier starting with a letter and followed only by letters, digits, and underscore characters, e.g `$id` . The simpler form is expanded by putting braces around the identifier, e.g `$id` is equivalent to `${id}` . In the following, unless we explicitly state otherwise, we assume that this expansion has already been performed.

The expanded expression is type checked normally. Usually, `StringContext` will resolve to the default implementation in the scala package, but it could also be user-defined. Note that new interpolators can also be added through implicit conversion of the built-in `scala.StringContext` .

One could write an extension

```
implicit class StringInterpolation(s: StringContext) {
  def id(args: Any*) = ???
}
```

Escape Sequences

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

charEscapeSeq	unicode	name	char
‘\‘ ‘b‘	\u0008	backspace	BS
‘\‘ ‘t‘	\u0009	horizontal tab	HT
‘\‘ ‘n‘	\u000a	linefeed	LF
‘\‘ ‘f‘	\u000c	form feed	FF
‘\‘ ‘r‘	\u000d	carriage return	CR
‘\‘ ‘"‘	\u0022	double quote	"
‘\‘ ‘'‘	\u0027	single quote	'
‘\‘ ‘\‘	\u005c	backslash	\

It is a compile time error if a backslash character in a character or string literal does not start a valid escape sequence.

Symbol literals

```
symbolLiteral  ::=  ''' plainid
```

A symbol literal `'x` is a shorthand for the expression `scala.Symbol("x")` and is of the literal type `'x. Symbol` is a case class, which is defined as follows.

```
package scala
final case class Symbol private (name: String) {
  override def toString: String = "'" + name
}
```

The `apply` method of `Symbol`'s companion object caches weak references to `Symbol`s, thus ensuring that identical symbol literals are equivalent with respect to reference equality.

Whitespace and Comments

Tokens may be separated by whitespace characters and/or comments. Comments come in two forms:

A single-line comment is a sequence of characters which starts with `//` and extends to the end of the line.

A multi-line comment is a sequence of characters between `/*` and `*/`. Multi-line comments may be nested, but are required to be properly nested. Therefore, a comment like `/* /* */` will be rejected as having an unterminated comment.

Trailing Commas in Multi-line Expressions

If a comma (`,`) is followed immediately, ignoring whitespace, by a newline and a closing parenthesis (`)`), bracket (`]`), or brace (`}`), then the comma is treated as a "trailing comma" and is ignored. For example:

```
foo(
  23,
  "bar",
  true,
)
```

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.

```
( whitespace | ‘(’ | ‘{’ ) ‘<’ (XNameStart | ‘!’ | ‘?’)

  XNameStart ::= ‘_’ | BaseChar | Ideographic // as in W3C XML, but withou
```

The scanner switches from XML mode to Scala mode if either of these is true:

- the XML expression or the XML pattern started by the initial ‘<’ has been successfully parsed
- the parser encounters an embedded Scala expression or pattern and forces the Scanner back to normal mode, until the Scala expression or pattern is successfully parsed. In this case, since code and XML fragments can be nested, the parser has to maintain a stack that reflects the nesting of XML and Scala expressions adequately

Note that no Scala tokens are constructed in XML mode, and that comments are interpreted as text.

Example

The following value definition uses an XML literal with two embedded Scala expressions:

```
val b = <book>
          <title>The Scala Language Specification</title>
          <version>{scalaBook.version}</version>
          <authors>{scalaBook.authors.mkList("", ", ", "")}</authors>
        </book>
```

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
}
```

A reference to an unqualified (type- or term-) identifier *x* is bound by the unique binding, which

- defines an entity with name *x* in the same namespace as the identifier
- shadows all other bindings that define entities with name *x* in that namespace.

It is an error if no such binding exists. If *x* is bound by an import clause, then the simple name *x* is taken to be equivalent to the qualified name to which *x* is mapped by the import clause. If *x* is bound by a definition or declaration, then *x* refers to the entity introduced by that binding. In that case, the type of *x* is the type of the referenced entity.

A reference to a qualified (type- or term-) identifier $e.x$ refers to the member of the type T of e which has the name x in the same namespace as the identifier. It is an error if T is not a value type. The type of $e.x$ is the member type of the referenced entity in T.

Binding precedence implies that the way source is bundled in files affects name resolution. In particular, imported names have higher precedence than names, defined in other files, that might otherwise be visible because they are defined in either the current package or an enclosing package.

Note that a package definition is taken as lowest precedence, since packages are open and can be defined across arbitrary compilation units.

```
package util {
  import scala.util
  class Random
  object Test extends App {
    println(new util.Random)  // scala.util.Random
  }
}
```

The compiler supplies imports in a preamble to every source file. This preamble conceptually has the following form, where braces indicate nested scopes:

```
import java.lang._
{
  import scala._
  {
    import Predef._
    { /* source */ }
  }
}
```

These imports are taken as lowest precedence, so that they are always shadowed by user code, which may contain competing imports and definitions. They also increase the nesting depth as shown, so that later imports shadow earlier ones.

As a convenience, multiple bindings of a type identifier to the same underlying type is permitted. This is possible when import clauses introduce a binding of a member type alias with the same binding precedence, typically through wildcard imports. This allows redundant type aliases to be imported without introducing an ambiguity.

```
object X { type T = annotation.tailrec }
object Y { type T = annotation.tailrec }
object Z {
  import X._, Y._, annotation.{tailrec => T}  // OK, all T mean tailrec
  @T def f: Int = { f ; 42 }                  // error, f is not tail recu
}
```

Similarly, imported aliases of names introduced by package statements are allowed, even though the names are strictly ambiguous:

```
// c.scala
package p { class C }

// xy.scala
import p._
package p { class X extends C }
package q { class Y extends C }
```

The reference to `C` in the definition of `X` is strictly ambiguous because `C` is available by virtue of the package clause in a different file, and can't shadow the imported name. But because the references are the same, the definition is taken as though it did shadow the import.

Example

Assume the following two definitions of objects named `X` in packages `p` and `q` in separate compilation units.

```
package p {
  object X { val x = 1; val y = 2 }
}

package q {
  object X { val x = true; val y = false }
}
```

The following program illustrates different kinds of bindings and precedences between them.

```
package p {                    // `X' bound by package clause
import Console._               // `println' bound by wildcard import
object Y {
  println(s"L4: $X")           // `X' refers to `p.X' here
  locally {
    import q._                 // `X' bound by wildcard import
    println(s"L7: $X")         // `X' refers to `q.X' here
    import X._                 // `x' and `y' bound by wildcard import
    println(s"L9: $x")         // `x' refers to `q.X.x' here
    locally {
      val x = 3                // `x' bound by local definition
      println(s"L12: $x")      // `x' refers to constant `3' here
      locally {
        import q.X._           // `x' and `y' bound by wildcard import
//      println(s"L15: $x")    // reference to `x' is ambiguous here
        import X.y             // `y' bound by explicit import
        println(s"L17: $y")    // `y' refers to `q.X.y' here
        locally {
          val x = "abc"        // `x' bound by local definition
          import p.X._         // `x' and `y' bound by wildcard import
//        println(s"L21: $y") // reference to `y' is ambiguous here
          println(s"L22: $x") // `x' refers to string "abc" here
}}}}}}
```

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

Classes and Objects

Expressions

TODO

Implicits

TODO

Pattern Matching

TODO

Top-Level Definitions

Top-Level Definitions

XML Expressions and Patterns

TODO

Annotations

TODO

The Scala Standard Library

TODO

Syntax Summary

TODO

References

TODO

Changelog

TODO