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
}}}}}}
```

Types

```
Type              ::=  FunctionArgTypes ‘=>’ Type
                    |  InfixType [ExistentialClause]
FunctionArgTypes  ::=  InfixType
                    |  ‘(’ [ ParamType {‘,’ ParamType } ] ‘)’
ExistentialClause ::=  ‘forSome’ ‘{’ ExistentialDcl
                          {semi ExistentialDcl} ‘}’
ExistentialDcl    ::=  ‘type’ TypeDcl
                    |  ‘val’ ValDcl
InfixType         ::=  CompoundType {id [nl] CompoundType}
CompoundType      ::=  AnnotType {‘with’ AnnotType} [Refinement]
                    |  Refinement
AnnotType         ::=  SimpleType {Annotation}
SimpleType        ::=  SimpleType TypeArgs
                    |  SimpleType ‘#’ id
                    |  StableId
                    |  Path ‘.’ ‘type’
                    |  Literal
                    |  ‘(’ Types ‘)’
TypeArgs          ::=  ‘[’ Types ‘]’
Types             ::=  Type {‘,’ Type}
```

We distinguish between first-order types and type constructors, which take type parameters and yield types. A subset of first-order types called *value types* represents sets of (first-class) values. Value types are either *concrete* or *abstract*.

Every concrete value type can be represented as a *class type*, i.e. a type designator that refers to a class or a trait [1], or as a compound type representing an intersection of types, possibly with a refinement that further constrains the types of its members.

Abstract value types are introduced by type parameters and abstract type bindings. Parentheses in types can be used for grouping.

Non-value types capture properties of identifiers that are not values. For example, a type constructor does not directly specify a type of values. However, when a type constructor is applied to the correct type arguments, it yields a first-order type, which may be a value type.

Non-value types are expressed indirectly in Scala. E.g., a method type is described by writing down a method signature, which in itself is not a real type, although it gives rise to a corresponding method type. Type constructors are another example, as one can write `type Swap[m[_, _], a,b] = m[b, a]`, but there is no syntax to write the corresponding anonymous type function directly.

[1] We assume that objects and packages also implicitly define a class (of the same name as the object or package, but inaccessible to user programs).

Paths

```
Path            ::=  StableId
                  |  [id '.'] this
StableId        ::=  id
                  |  Path '.' id
                  |  [id '.'] 'super' [ClassQualifier] '.' id
ClassQualifier  ::= '[' id ']'
```

Paths are not types themselves, but they can be a part of named types and in that function form a central role in Scala's type system.

A path is one of the following.

- The empty path # (which cannot be written explicitly in user programs).
- C`.this`, where C references a class. The path `this` is taken as a shorthand for C`.this` where C is the name of the class directly enclosing the reference.
- $p.x$ where p is a path and x is a stable member of p. *Stable members* are packages or members introduced by object definitions or by value definitions of non-volatile types.
- C`.super`x or C`.super`$[M].x$ where C references a class and x references a stable member of the super class or designated parent class M of C. The prefix `super` is taken as a shorthand for C`.super` where C is the name of the class directly enclosing the reference.

A *stable identifier* is a path which ends in an identifier.

Value Types

Every value in Scala has a type which is of one of the following forms.

Singleton Types

```
SimpleType  ::=  Path '.' 'type'
```

A *singleton type* is of the form $p.$`type`. Where p is a path pointing to a value which conforms to `scala.AnyRef`, the type denotes the set of values consisting of `null` and the value denoted by p (i.e., the value v for which `v eq p`). Where the path does not conform to `scala.AnyRef` the type denotes the set consisting of only the value denoted by p.

Literal Types

```
SimpleType  ::=  Literal
```

A literal type `lit` is a special kind of singleton type which denotes the single literal value `lit`. Thus, the type ascription `1: 1` gives the most precise type to the literal value `1`: the literal type `1`.

At run time, an expression `e` is considered to have literal type `lit` if `e == lit`. Concretely, the result of `e.isInstanceOf[lit]` and `e match { case _ : lit => }` is determined by evaluating `e == lit`.

Literal types are available for all types for which there is dedicated syntax, except `Unit`. This includes the numeric types (other than `Byte` and `Short` which don't currently have syntax), `Boolean`, `Char`, `String` and `Symbol`.

Stable Types

A *stable type* is a singleton type, a literal type, or a type that is declared to be a subtype of trait `scala.Singleton`.

Type Projection

```
SimpleType  ::=  SimpleType ‘#’ id
```

A *type projection* $T\#x$ references the type member named x of type T.

Type Designators

```
SimpleType  ::=  StableId
```

A *type designator* refers to a named value type. It can be simple or qualified. All such type designators are shorthands for type projections.

Specifically, the unqualified type name t where t is bound in some class, object, or package C is taken as a shorthand for C`.this.type#`t. If t is not bound in a class, object, or package, then t is taken as a shorthand for `#.type#`t.

A qualified type designator has the form `p.t` where `p` is a path and t is a type name. Such a type designator is equivalent to the type projection `p.type#t`.

Example
Some type designators and their expansions are listed below. We assume a local type parameter t, a value `maintable` with a type member `Node` and the standard class `scala.Int`,

Designator	Expansion
`t`	`#.type#t`
`Int`	`scala.type#Int`
`scala.Int`	`scala.type#Int`

```
data.maintable.Node             data.maintable.type#Node
```

Parameterized Types

```
SimpleType      ::=  SimpleType TypeArgs
TypeArgs        ::=  '[' Types ']'
```

A *parameterized type* $T[T_1, \ldots, T_n]$ consists of a type designator T and type parameters $T_1, \ldots, T_n$ where $n \geq 1$.
T must refer to a type constructor which takes n type parameters $a_1, \ldots, a_n$.

Say the type parameters have lower bounds $L_1, \ldots, L_n$ and upper bounds $U_1, \ldots, U_n$. The parameterized type is well-formed if each actual type parameter *conforms to its bounds*, i.e. $\sigma L_i \<: T_i \<: \sigma U_i$ where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$.

Example
Given the partial type definitions:

```
class TreeMap[A <: Comparable[A], B] { … }
class List[A] { … }
class I extends Comparable[I] { … }

class F[M[_], X] { … }
class S[K <: String] { … }
class G[M[ Z <: I ], I] { … }
```

the following parameterized types are well formed:

```
TreeMap[I, String]
List[I]
List[List[Boolean]]

F[List, Int]
G[S, String]
```

Example
Given the above type definitions, the following types are ill-formed:

```
TreeMap[I]            // illegal: wrong number of parameters
TreeMap[List[I], Int] // illegal: type parameter not within bound
```

```
F[Int, Boolean]       // illegal: Int is not a type constructor
F[TreeMap, Int]       // illegal: TreeMap takes two parameters,
                      //   F expects a constructor taking one
G[S, Int]             // illegal: S constrains its parameter to
                      //   conform to String,
                      // G expects type constructor with a parameter
                      //   that conforms to Int
```

Tuple Types

```
SimpleType    ::=   '(' Types ')'
```

A *tuple type* $(T_1, \ldots, T_n)$ is an alias for the class `scala.Tuple`$_n$`[`T_1`, … , `T_n`]`, where $n \geq 2$.

Tuple classes are case classes whose fields can be accessed using selectors `_1` , … , `_n` . Their functionality is abstracted in a corresponding `Product` trait. The n-ary tuple class and product trait are defined at least as follows in the standard Scala library (they might also add other methods and implement other traits).

```
case class Tuple_n[+T_1, … , +T_n](_1: T_1, … , _n: T_n)
extends Product_n[T_1, … , T_n]

trait Product_n[+T_1, … , +T_n] {
  override def productArity = n
  def _1: T_1
  …
  def _n: T_n
}
```

Annotated Types

```
AnnotType  ::=  SimpleType {Annotation}
```

An *annotated type* $T\ a_1, \ldots, a_n$ attaches annotations $a_1, \ldots, a_n$ to the type T.

Example
The following type adds the `@suspendable` annotation to the type `String` :

```
String @suspendable
```

Compound Types

```
CompoundType    ::=  AnnotType {‘with’ AnnotType} [Refinement]
                  |  Refinement
Refinement      ::=  [nl] ‘{’ RefineStat {semi RefineStat} ‘}’
RefineStat      ::=  Dcl
                  |  ‘type’ TypeDef
                  |
```

A *compound type* T_1 `with` ... `with` $T_n\{R\}$ represents objects with members as given in the component types $T_1, \ldots, T_n$ and the refinement $\{R\}$. A refinement $\{R\}$ contains declarations and type definitions. If a declaration or definition overrides a declaration or definition in one of the component types $T_1, \ldots, T_n$, the usual rules for overriding apply; otherwise the declaration or definition is said to be “structural” [2].

Within a method declaration in a structural refinement, the type of any value parameter may only refer to type parameters or abstract types that are contained inside the refinement. That is, it must refer either to a type parameter of the method itself, or to a type definition within the refinement. This restriction does not apply to the method's result type.

If no refinement is given, the empty refinement is implicitly added, i.e. T_1 `with` ... `with` T_n is a shorthand for T_1 `with` ... `with` $T_n\{\}$.

A compound type may also consist of just a refinement $\{R\}$ with no preceding component types. Such a type is equivalent to `AnyRef` $\{R\}$.

Example
The following example shows how to declare and use a method which has a parameter type that contains a refinement with structural declarations.

```
case class Bird (val name: String) extends Object {
  def fly(height: Int) = …
}
case class Plane (val callsign: String) extends Object {
  def fly(height: Int) = …
}
def takeoff(
  runway: Int,
  r: { val callsign: String; def fly(height: Int) }
) = {
  tower.print(r.callsign + " requests take-off on runway " + runway)
  tower.read(r.callsign + " is clear for take-off")
  r.fly(1000)
}
val bird = new Bird("Polly the parrot"){ val callsign = name }
val a380 = new Plane("TZ-987")
```

```
takeoff(42, bird)
takeoff(89, a380)
```

Although `Bird` and `Plane` do not share any parent class other than `Object`, the parameter *r* of method `takeoff` is defined using a refinement with structural declarations to accept any object that declares a value `callsign` and a `fly` method.

2 A reference to a structurally defined member (method call or access to a value or variable) may generate binary code that is significantly slower than an equivalent code to a non-structural member.

Infix Types

```
InfixType     ::=  CompoundType {id [nl] CompoundType}
```

An *infix type* T_1 `op` T_2 consists of an infix operator `op` which gets applied to two type operands T_1 and T_2. The type is equivalent to the type application `op`$[T_1, T_2]$. The infix operator `op` may be an arbitrary identifier.

All type infix operators have the same precedence; parentheses have to be used for grouping. The associativity of a type operator is determined as for term operators: type operators ending in a colon ':' are right-associative; all other operators are left-associative.

In a sequence of consecutive type infix operations $t_0, op, t_1, op_2, \ldots, op_n, t_n$, all operators $op_1, \ldots, op_n$ must have the same associativity. If they are all left-associative, the sequence is interpreted as $(\ldots(t_0 op_1 t_1) op_2 \ldots) op_n t_n$, otherwise it is interpreted as $t_0 op_1 (t_1 op_2 (\ldots op_n t_n) \ldots)$.)

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 $T_1, \ldots, Tn$ 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 Function_n[-T_1 , ... , -T_n, ] {
```

```
  def apply(x1: T1 , ... , xn: Tn): U
  override def toString = "<function>"
}
```

Hence, function types are covariant in their result type and contravariant in their argument types.

Existential Types

```
Type               ::= InfixType ExistentialClauses
ExistentialClauses ::= 'forSome' '{' ExistentialDcl
                       {semi ExistentialDcl} '}'
ExistentialDcl     ::= 'type' TypeDcl
                     |  'val' ValDcl
```

An *existential type* has the form T `forSome { `Q` }` where Q is a sequence of type declarations.

Let $t_1[tps_1] >: L_1 <: U_1, \ldots, t_n[tps_n] >: L_n <: U_n$ be the types declared in Q (any of the type parameter sections `[ `tps_i` ]` might be missing). The scope of each type t_i includes the type T and the existential clause Q. The type variables t_i are said to be *bound* in the type T `forSome { `Q` }`. Type variables which occur in a type T but which are not bound in T are said to be *free* in T.

A *type instance* of T `forSome { `Q` }` is a type σT where σ is a substitution over $t_1, \ldots, t_n$ such that, for each i, $\sigma L_i <: \sigma t_i <: \sigma U_i$. The set of values denoted by the existential type T `forSome {`Q`}` is the union of the set of values of all its type instances.

A *skolemization* of T `forSome { `Q` }` is a type instance σT, where σ is the substitution $[t'_1/t_1, \ldots, t'_n/t_n]$ and each t'_i is a fresh abstract type with lower bound σL_i and upper bound σU_i.

Simplification Rules

Existential types obey the following four equivalences:

1. Multiple for-clauses in an existential type can be merged.
   E.g., T `forSome { `Q` } forSome { `Q'` }` is equivalent to T `forSome { `Q` ; `Q'`}`.
2. Unused quantifications can be dropped.
   E.g., T `forSome { `Q` ; `Q'`}` where none of the types defined in Q' are referred to by T or Q, is equivalent to T `forSome {`Q`}`.
3. An empty quantification can be dropped. E.g., T `forSome { }` is equivalent to T.
4. An existential type T `forSome { `Q` }` where Q contains a clause `type` $t[tps] >: L <: U$ is equivalent to the type T' `forSome { `Q` }` where T' results from T by replacing every covariant occurrence of t in T by U and by replacing every contravariant occurrence of t in T by L.

Existential Quantification over Values

As a syntactic convenience, the bindings clause in an existential type may also contain value declarations `val` x`:` T. An existential type T `forSome { ` Q`; val` x`:` S`;` Q' `}` is treated as a shorthand for the type T' `forSome { ` Q`; type` t `<:` S `with Singleton; ` Q' `}`, where t is a fresh type name and T' results from T by replacing every occurrence of x`.type` with t.

Placeholder Syntax for Existential Types

```
WildcardType   ::=  '_' TypeBounds
```

Scala supports a placeholder syntax for existential types. A *wildcard type* is of the form `_>:`L`<:`U. Both bound clauses may be omitted. If a lower bound clause `>:`L is missing, `>:scala.Nothing` is assumed. If an upper bound clause `<:`U is missing, `<:scala.Any` is assumed.
A wildcard type is a shorthand for an existentially quantified type variable, where the existential quantification is implicit.

A wildcard type must appear as type argument of a parameterized type. Let $T = p.c[targs, T, targs']$ be a parameterized type where $targs, targs'$ may be empty and T is a wildcard type `_>:`L`<:`U. Then T is equivalent to the existential type

$p.c[targs, t, targs']$ `forSome { type` t `>:` L `<:` U `}`

where t is some fresh type variable. Wildcard types may also appear as parts of infix types, function types, or tuple types. Their expansion is then the expansion in the equivalent parameterized type.

Example
Assume the class definitions

```
class Ref[T]
abstract class Outer { type T }
```

Here are some examples of existential types:

```
Ref[T] forSome { type T <: java.lang.Number }
Ref[x.T] forSome { val x: Outer }
Ref[x_type # T] forSome { type x_type <: Outer with Singleton }
```

The last two types in this list are equivalent. An alternative formulation of the first type above using wildcard syntax is:

```
Ref[_ <: java.lang.Number]
```

Example

The type `List[List[_]]` is equivalent to the existential type

```
List[List[t] forSome { type t }]
```

Example
Assume a covariant type

```
class List[+T]
```

The type

```
List[T] forSome { type T <: java.lang.Number }
```

is equivalent (by simplification rule 4 above) to

```
List[java.lang.Number] forSome { type T <: java.lang.Number }
```

which is in turn equivalent (by simplification rules 2 and 3 above) to `List[java.lang.Number]`.

Non-Value Types

The types explained in the following do not denote sets of values, nor do they appear explicitly in programs. They are introduced in this report as the internal types of defined identifiers.

Method Types

A *method type* is denoted internally as $(Ps)U$, where (Ps) is a sequence of parameter names and types $(p_1:T_1, \ldots, p_n:T_n)$ for some $n \geq 0$ and U is a (value or method) type.
This type represents named methods that take arguments named $p_1, \ldots, p_n$ of types $T_1, \ldots, T_n$ and that return a result of type U.

Method types associate to the right: $(Ps_1)(Ps_2)U$ is treated as $(Ps_1)((Ps_2)U)$.

A special case are types of methods without any parameters. They are written here `=> T`.
Parameterless methods name expressions that are re-evaluated each time the parameterless method name is referenced.

Method types do not exist as types of values. If a method name is used as a value, its type is implicitly converted to a corresponding function type.

Example
The declarations

```
def a: Int
def b (x: Int): Boolean
def c (x: Int) (y: String, z: String): String
```

produce the typings

```
a: => Int
b: (Int) Boolean
c: (Int) (String, String) String
```

Polymorphic Method Types

A polymorphic method type is denoted internally as $[tps]T$ where $[tps]$ is a type parameter section $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n]$ for some $n \geq 0$ and T is a (value or method) type. This type represents named methods that take type arguments $S_1, \ldots, S_n$ which conform to the lower bounds $L_1, \ldots, L_n$ and the upper bounds $U_1, \ldots, U_n$ and that yield results of type T.

Example
The declarations

```
def empty[A]: List[A]
def union[A <: Comparable[A]] (x: Set[A], xs: Set[A]): Set[A]
```

produce the typings

```
empty : [A >: Nothing <: Any] List[A]
union : [A >: Nothing <: Comparable[A]] (x: Set[A], xs: Set[A]) Set[A]
```

Type Constructors

A *type constructor* is represented internally much like a polymorphic method type. $[\pm a_1 >: L_1 <: U_1, \ldots, \pm a_n >: L_n <: U_n] T$ represents a type that is expected by a type constructor parameter or an abstract type constructor binding with the corresponding type parameter clause.

Example
Consider this fragment of the `Iterable[+X]` class:

```
trait Iterable[+X] {
  def flatMap[newType[+X] <: Iterable[X], S](f: X => newType[S]): newTy
}
```

> Conceptually, the type constructor `Iterable` is a name for the anonymous type `[+X] Iterable[X]`, which may be passed to the `newType` type constructor parameter in `flatMap`.

Base Types and Member Definitions

Types of class members depend on the way the members are referenced. Central here are three notions, namely:

1. the notion of the set of base types of a type T
2. the notion of a type T in some class C seen from some prefix type S
3. the notion of the set of member bindings of some type T

These notions are defined mutually recursively as follows.

1. The set of *base types* of a type is a set of class types, given as follows.
   - The base types of a class type C with parents $T_1, \ldots, T_n$ are C itself, as well as the base types of the compound type T_1 `with` … `with` T_n `{` R `}`.
   - The base types of an aliased type are the base types of its alias.
   - The base types of an abstract type are the base types of its upper bound.
   - The base types of a parameterized type C`[`$T_1, \ldots, T_n$`]` are the base types of type C, where every occurrence of a type parameter a_i of C has been replaced by the corresponding parameter type T_i.
   - The base types of a singleton type p`.type` are the base types of the type of p.
   - The base types of a compound type T_1 `with` … `with` T_n `{` R `}` are the *reduced union* of the base classes of all T_i's. This means: Let the multi-set S be the multi-set-union of the base types of all T_i's. If S contains several type instances of the same class, say S^i`#`C`[`$T^i_1, \ldots, T^i_n$`]` $(i \in I)$, then all those instances are replaced by one of them which conforms to all others. It is an error if no such instance exists. It follows that the reduced union, if it exists, produces a set of class types, where different types are instances of different classes.
   - The base types of a type selection S`#`T are determined as follows. If T is an alias or abstract type, the previous clauses apply. Otherwise, T must be a (possibly parameterized) class type, which is defined in some class B. Then the base types of S`#`T are the base types of T in B seen from the prefix type S.
   - The base types of an existential type T `forSome {` Q `}` are all types S `forSome {` Q `}` where S is a base type of T.
2. The notion of a type T *in class* C *seen from some prefix type* S makes sense only if the prefix type S has a type instance of class C as a base type, say S'`#`C`[`$T_1, \ldots, T_n$`]`. Then we define as follows.
   - If S `=` ϵ`.type`, then T in C seen from S is T itself.
   - Otherwise, if S is an existential type S' `forSome {` Q `}`, and T in C seen from S' is T', then T in C seen from S is T' `forSome {`Q`}`.
   - Otherwise, if T is the i'th type parameter of some class D, then

- If S has a base type $D[U_1, \ldots, U_n]$, for some type parameters $[U_1, \ldots, U_n]$, then T in C seen from S is U_i.
- Otherwise, if C is defined in a class C', then T in C seen from S is the same as T in C' seen from S'.
- Otherwise, if C is not defined in another class, then T in C seen from S is T itself.

- Otherwise, if T is the singleton type D`.this.type` for some class D then
  - If D is a subclass of C and S has a type instance of class D among its base types, then T in C seen from S is S.
  - Otherwise, if C is defined in a class C', then T in C seen from S is the same as T in C' seen from S'.
  - Otherwise, if C is not defined in another class, then T in C seen from S is T itself.
- If T is some other type, then the described mapping is performed to all its type components.

If T is a possibly parameterized class type, where T's class is defined in some other class D, and S is some prefix type, then we use "T seen from S" as a shorthand for "T in D seen from S".

3. The *member bindings* of a type T are
   1. all bindings d such that there exists a type instance of some class C among the base types of T and there exists a definition or declaration d' in C such that d results from d' by replacing every type T' in d' by T' in C seen from T
   2. all bindings of the type's refinement, if it has one

The *definition* of a type projection `S#T` is the member binding d_T of the type `T` in `S`. In that case, we also say that `S#T` *is defined by* d_T.

Relations between types

We define the following relations between types.

Name	Symbolically	Interpretation
Equivalence	$T \equiv U$	T and U are interchangeable in all contexts.
Conformance	$T <: U$	Type T conforms to (" is a subtype of") type U.
Weak Conformance	$T <:_w U$	Augments conformance for primitive numeric types.
Compatibility		Type T conforms to type U after conversions.

Equivalence

Equivalence $(\equiv)$ between types is the smallest congruence [3] such that the following holds:

- If t is defined by a type alias `type` t `=` T, then t is equivalent to T.
- If a path p has a singleton type q`.type`, then p`.type` $\equiv q$`.type`.

- If O is defined by an object definition, and p is a path consisting only of package or object selectors and ending in O, then O`.this.type` $\equiv p$`.type`.
- Two compound types are equivalent if the sequences of their component are pairwise equivalent, and occur in the same order, and their refinements are equivalent. Two refinements are equivalent if they bind the same names and the modifiers, types and bounds of every declared entity are equivalent in both refinements.
- Two method types are equivalent if:
  - neither are implicit, or they both are [4];
  - they have equivalent result types;
  - they have the same number of parameters; and
  - corresponding parameters have equivalent types. Note that the names of parameters do not matter for method type equivalence.
- Two polymorphic method types are equivalent if they have the same number of type parameters, and, after renaming one set of type parameters by another, the result types as well as lower and upper bounds of corresponding type parameters are equivalent.
- Two existential types are equivalent if they have the same number of quantifiers, and, after renaming one list of type quantifiers by another, the quantified types as well as lower and upper bounds of corresponding quantifiers are equivalent.
- Two type constructors are equivalent if they have the same number of type parameters, and, after renaming one list of type parameters by another, the result types as well as variances, lower and upper bounds of corresponding type parameters are equivalent.

[3] A congruence is an equivalence relation which is closed under formation of contexts.

[4] A method type is implicit if the parameter section that defines it starts with the `implicit` keyword.

Conformance

The conformance relation ($<:$) is the smallest transitive relation that satisfies the following conditions.

- Conformance includes equivalence. If $T \equiv U$ then $T <: U$.
- For every value type T, `scala.Nothing <:` T `<: scala.Any`.
- For every type constructor T (with any number of type parameters), `scala.Nothing <:` T `<: scala.Any`.
- For every class type T such that T `<: scala.AnyRef` one has `scala.Null <:` T.
- A type variable or abstract type t conforms to its upper bound and its lower bound conforms to t.
- A class type or parameterized type conforms to any of its base-types.
- A singleton type p`.type` conforms to the type of the path p.
- A singleton type p`.type` conforms to the type `scala.Singleton`.
- A type projection T`#`t conforms to U`#`t if T conforms to U.
- A parameterized type T`[`T_1`, … ,` T_n`]` conforms to T`[`U_1`, … ,` U_n`]` if the following three conditions hold for $i \in 1, \ldots, n$:
  1. If the i'th type parameter of T is declared covariant, then $T_i <: U_i$.
  2. If the i'th type parameter of T is declared contravariant, then $U_i <: T_i$.
  3. If the i'th type parameter of T is declared neither covariant nor contravariant, then $U_i \equiv T_i$.
- A compound type T_1 `with … with` T_n `{`R`}` conforms to each of its component types T_i.

- If $T <: U_i$ for $i \in 1, \ldots, n$ and for every binding d of a type or value x in R there exists a member binding of x in T which subsumes d, then T conforms to the compound type U_1 `with` ... `with` U_n `{`R`}`.
- The existential type T `forSome {`Q`}` conforms to U if its skolemization conforms to U.
- The type T conforms to the existential type U `forSome {`Q`}` if T conforms to one of the type instances of U `forSome {`Q`}`.
- If $T_i \equiv T'_i$ for $i \in 1, \ldots, n$ and U conforms to U' then the method type $(p_1 : T_1, \ldots, p_n : T_n)U$ conforms to $(p'_1 : T'_1, \ldots, p'_n : T'_n)U'$.
- The polymorphic type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n]T$ conforms to the polymorphic type $[a_1 >: L'_1 <: U'_1, \ldots, a_n >: L'_n <: U'_n]T'$ if, assuming $L'_1 <: a_1 <: U'_1, \ldots, L'_n <: a_n <: U'_n$ one has $T <: T'$ and $L_i <: L'_i$ and $U'_i <: U_i$ for $i \in 1, \ldots, n$.
- Type constructors T and T' follow a similar discipline. We characterize T and T' by their type parameter clauses $[a_1, \ldots, a_n]$ and $[a'_1, \ldots, a'_n]$, where an a_i or a'_i may include a variance annotation, a higher-order type parameter clause, and bounds. Then, T conforms to T' if any list $[t_1, \ldots, t_n]$ -- with declared variances, bounds and higher-order type parameter clauses -- of valid type arguments for T' is also a valid list of type arguments for T and $T[t_1, \ldots, t_n] <: T'[t_1, \ldots, t_n]$. Note that this entails that:
  - The bounds on a_i must be weaker than the corresponding bounds declared for a'_i.
  - The variance of a_i must match the variance of a'_i, where covariance matches covariance, contravariance matches contravariance and any variance matches invariance.
  - Recursively, these restrictions apply to the corresponding higher-order type parameter clauses of a_i and a'_i.

A declaration or definition in some compound type of class type C *subsumes* another declaration of the same name in some compound type or class type C', if one of the following holds.

- A value declaration or definition that defines a name x with type T subsumes a value or method declaration that defines x with type T', provided $T <: T'$.
- A method declaration or definition that defines a name x with type T subsumes a method declaration that defines x with type T', provided $T <: T'$.
- A type alias `type` t`[`T_1 `,` ... `,` T_n`]` `=` T subsumes a type alias `type` t`[`T_1 `,` ... `,` T_n`]` `=` T' if $T \equiv T'$.
- A type declaration `type` t`[`T_1 `,` ... `,` T_n`]` `>:` L `<:` U subsumes a type declaration `type` t`[`T_1 `,` ... `,` T_n`]` `>:` L' `<:` U' if $L' <: L$ and $U <: U'$.
- A type or class definition that binds a type name t subsumes an abstract type declaration `type` `t[`T_1 `,` ... `,` T_n`]` `>:` `L` `<:` `U` if $L <: t <: U$.

Least upper bounds and greatest lower bounds

The $(<:)$ relation forms pre-order between types, i.e. it is transitive and reflexive. This allows us to define *least upper bounds* and *greatest lower bounds* of a set of types in terms of that order. The least upper bound or greatest lower bound of a set of types does not always exist. For instance, consider the class definitions:

```
class A[+T] {}
class B extends A[B]
class C extends A[C]
```

Then the types `A[Any], A[A[Any]], A[A[A[Any]]], ...` form a descending sequence of upper bounds for `B` and `C`. The least upper bound would be the infinite limit of that sequence, which does not exist as a Scala type. Since cases like this are in general impossible to detect, a Scala compiler is free to reject a term which has a type specified as a least upper or greatest lower bound, and that bound would be more complex than some compiler-set limit [^4].

The least upper bound or greatest lower bound might also not be unique. For instance `A with B` and `B with A` are both greatest lower bounds of `A` and `B`. If there are several least upper bounds or greatest lower bounds, the Scala compiler is free to pick any one of them.

[^4]: The current Scala compiler limits the nesting level of parameterization in such bounds to be at most two deeper than the maximum nesting level of the operand types

Weak Conformance

In some situations Scala uses a more general conformance relation. A type S *weakly conforms* to a type T, written $S \<:_w T$, if $S \<: T$ or both S and T are primitive number types and S precedes T in the following ordering.

```
Byte   Short
Short  Int
Char   Int
Int    Long
Long   Float
Float  Double
```

A *weak least upper bound* is a least upper bound with respect to weak conformance.

Compatibility

A type T is *compatible* to a type U if T (or its corresponding function type) weakly conforms to U after applying eta-expansion. If T is a method type, it's converted to the corresponding function type. If the types do not weakly conform, the following alternatives are checked in order:

- view application: there's an implicit view from T to U
- dropping by-name modifiers: if U is of the shape $= \> U'$ (and T is not), $T \<:_w U'$
- SAM conversion: if T corresponds to a function type, and U declares a single abstract method whose type corresponds to the function type U', $T \<:_w U'$

Example

Function compatibility via SAM conversion

Given the definitions

```
def foo(x: Int => String): Unit
def foo(x: ToString): Unit

trait ToString { def convert(x: Int): String }
```

The application `foo((x: Int) => x.toString)` resolves to the first overload, as it's more specific:

- `Int => String` is compatible to `ToString` -- when expecting a value of type `ToString`, you may pass a function literal from `Int` to `String`, as it will be SAM-converted to said function;
- `ToString` is not compatible to `Int => String` -- when expecting a function from `Int` to `String`, you may not pass a `ToString`.

Volatile Types

Type volatility approximates the possibility that a type parameter or abstract type instance of a type does not have any non-null values. A value member of a volatile type cannot appear in a path.

A type is *volatile* if it falls into one of four categories:

A compound type T_1 `with` … `with` T_n `{`R`}` is volatile if one of the following two conditions hold.

1. One of $T_2, \ldots, T_n$ is a type parameter or abstract type, or
2. T_1 is an abstract type and and either the refinement R or a type T_j for j > 1 contributes an abstract member to the compound type, or
3. one of $T_1, \ldots, T_n$ is a singleton type.

Here, a type S *contributes an abstract member* to a type T if S contains an abstract member that is also a member of T. A refinement R contributes an abstract member to a type T if R contains an abstract declaration which is also a member of T.

A type designator is volatile if it is an alias of a volatile type, or if it designates a type parameter or abstract type that has a volatile type as its upper bound.

A singleton type p`.type` is volatile, if the underlying type of path p is volatile.

An existential type T `forSome` `{`Q`}` is volatile if T is volatile.

Type Erasure

A type is called *generic* if it contains type arguments or type variables. *Type erasure* is a mapping from (possibly generic) types to non-generic types. We write $|T|$ for the erasure of type T. The erasure mapping is defined as follows.

- The erasure of an alias type is the erasure of its right-hand side.
- The erasure of an abstract type is the erasure of its upper bound.
- The erasure of the parameterized type `scala.Array`$[T_1]$ is `scala.Array`$[|T_1|]$.
- The erasure of every other parameterized type $T[T_1, \ldots, T_n]$ is $|T|$.

- The erasure of a singleton type p`.type` is the erasure of the type of p.
- The erasure of a type projection T`#`x is $|T|$`#`x.
- The erasure of a compound type T_1 `with` ... `with` T_n `{`R`}` is the erasure of the intersection dominator of $T_1, \ldots, T_n$.
- The erasure of an existential type T `forSome {`Q`}` is $|T|$.

The *intersection dominator* of a list of types $T_1, \ldots, T_n$ is computed as follows. Let $T_{i_1}, \ldots, T_{i_m}$ be the subsequence of types T_i which are not supertypes of some other type T_j. If this subsequence contains a type designator T_c that refers to a class which is not a trait, the intersection dominator is T_c. Otherwise, the intersection dominator is the first element of the subsequence, T_{i_1}.

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