Lexical Syntax

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 '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`), modifier letters (`Ml`), 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 | charEscapeSeq } ‘`’
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 or `_`, and *constant identifiers*, which do not.

For this purpose, lower case letter don't only include a-z, but also all characters in Unicode category Ll (lowercase letter), as well as all letters that have contributory property Other_Lowercase, except characters in category Nl (letter numerals) which are never taken as lower case.

The following are examples of variable identifiers:

```
x          maxIndex   p2p   empty_?
`yield`           _y    dot_product_*
__system   _MAX_LEN_
ªpple      helper
```

Some examples of constant identifiers are

```
+    Object  $reserved  Džul     nûm
i_iii  I_III      elerious   qhàà   thatsaletter
```

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.

Here are examples of identifiers:

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

When one needs to access Java identifiers that are reserved words in Scala, use backquote-enclosed strings. For instance, the statement `Thread.yield()` is illegal, since `yield` is a reserved word in Scala. However, here's a work-around: ``Thread.`yield`()``

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, and
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, and
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, and
- 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, and
- after an annotation.

  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]
```

```
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 }
}
```

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

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

```
  x < 0 ||

  x > 10
```

```
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
```

```
@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 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’
```

Values of type `Int` are all integer numbers between -231 and $231-1$, inclusive. Values of type `Long` are all integer numbers between -263 and $263-1$, inclusive. A compile-time error occurs if an integer literal denotes a number outside these ranges.

Integer literals are usually of type `Int`, or of type `Long` when followed by a `L` or `l` suffix. (Lowercase `l` is deprecated for reasons of legibility.)

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`	-27 to $27-1$
`Short`	-215 to $215-1$
`Char`	0 to $216-1$

The digits of a numeric literal may be separated by arbitrarily many underscores for purposes of legibility.

```
0            21_000       0x7F         -42L         0xFFFF_FFFF
```

Floating Point Literals

```
floatingPointLiteral  ::= digit {digit} ‘.’ digit {digit} [exponentPart] [floatType]
                        |  ‘.’ 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.

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

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

`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 | charEscapeSeq) ‘'’
```

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.

```
'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 | charEscapeSeq
```

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`.

```
"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.

```
  """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.StringOps.

Interpolated string

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

Interpolated string consist 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 a 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 is 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

```scala
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 without ‘:’
```

The scanner switches from XML mode to Scala mode if either

- the XML expression or the XML pattern started by the initial ‘<’ has been successfully parsed, or if
- 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.

> 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>
> ```