The Scala Language Specification, Version 2.9

24th May 2011

Contents

0.1 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 two less pure but mainstream object-oriented languages – Java and C#.

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 (an alternative implementation of Scala also works for .NET). 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 (Odersky and al. 2004; Odersky 2006; Odersky and Zenger 2005a; Odersky et al. 2003; Odersky and Zenger 2005b)

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.

Chapter 1

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), letter numerals(Nl) and the two characters \u0024 '\$' and \u005F '_', which both count as upper case letters
3. Digits `‘0’ | … | ‘9’`
4. Parentheses `‘(’ | ‘)’ | ‘[’ | ‘]’ | ‘{’ | ‘}’`
5. Delimiter characters `“’ | ‘” | ‘”’ | ‘.’ | ‘;’ | ‘,’`
6. Operator characters. These consist of all printable ASCII characters \u0020-\u007F which are in none of the sets above, mathematical symbols(Sm) and other symbols(So).

1.1 Identifiers

```
op       ::=  opchar {opchar}
```

```
varid    ::=  lower idrest
plainid  ::=  upper idrest
           |  varid
           |  op
id       ::=  plainid
           |  ‘‘’ stringLit ‘‘’
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.

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        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.

(1) Here are examples of identifiers:

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

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

1.2 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 an conditional expression (here) or while loop (here) and the next following expression,
- between the enumerators of a for-comprehension (here) and the next following expression, and
- after the initial **`type`** keyword in a type definition or declaration (here).

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 (here),
- in front of a parameter clause (here), and
- after an annotation (here).

(3) The following code contains four well-formed statements, each on two lines. 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]
```

(4) The following code designates an anonymous class:

```
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 }
}
```

(5) The following code designates a single expression:

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

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

```
  x < 0 ||

  x > 10
```

(6) The following code designates a single, curried function definition:

```
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
```

(7) The following code designates an attributed definition:

```
@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 { ... }
```

1.3 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
           |  symbolLiteral
           |  'null'
```

1.3.1 Integer Literals

```
integerLiteral  ::= (decimalNumeral | hexNumeral | octalNumeral) ['L' | 'l']
decimalNumeral   ::=  '0' | nonZeroDigit {digit}
hexNumeral       ::=  '0' 'x' hexDigit {hexDigit}
octalNumeral     ::=  '0' octalDigit {octalDigit}
digit            ::=  '0' | nonZeroDigit
nonZeroDigit     ::=  '1' | … | '9'
octalDigit       ::=  '0' | … | '7'
```

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 $2^{16}-1$

(8) Here are some integer literals:

```
0          21          0xFFFFFFFF       0777L
```

1.3.2 Floating Point Literals

```
floatingPointLiteral ::= 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.

(9) Here are some floating point literals:

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

(10) The phrase `1.toString` parses as three different tokens: `1`, `.`, and `toString`. On the other hand, if a space is inserted after the period, the phrase `1. toString` parses as the floating point literal `1.` followed by the identifier `toString`.

1.3.3 Boolean Literals

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

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

1.3.4 Character Literals

```
characterLiteral  ::=  ‘'’ printableChar ‘'’
                    |  ‘'’ charEscapeSeq ‘'’
```

A character literal is a single character enclosed in quotes. The character is either a printable unicode character or is described by an escape sequence.

(11) Here are some character literals:

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

Note that `'\u000A'` is *not* a valid character literal because Unicode conversion is done before literal parsing and the Unicode character \u000A (line feed) is not a printable character. One can use instead the escape sequence `'\n'` or the octal escape `'\12'` (see here).

1.3.5 String Literals

```
stringLiteral  ::=  ‘\"’ {stringElement} ‘\"’
stringElement  ::=  printableCharNoDoubleQuote  |  charEscapeSeq
```

A string literal is a sequence of characters in double quotes. The characters are either printable unicode character or are described by escape sequences. If the string literal contains a double quote character, it must be escaped, i.e. "\"". The value of a string literal is an instance of class `String`.

(12) Here are some string literals:

```
"Hello,\nWorld!"
"This string contains a \" character."
```

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 consuctive 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.

(13) Here is a multi-line string literal:

```
"""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.

1.3.6 Escape Sequences

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

`\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 \

A character with Unicode between 0 and 255 may also be represented by an octal escape, i.e. a backslash '' followed by a sequence of up to three octal characters.

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

1.3.7 Symbol literals

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

A symbol literal `'x` is a shorthand for the expression `scala.Symbol("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.

1.4 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.

1.5 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.

(14) 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>
```

Chapter 2

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 in the same compilation unit where the definition occurs 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 not in the compilation unit where the definition occurs 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 a situation like

```
val x = 1;
{ import p.x;
  x }
```

neither binding of x shadows the other. Consequently, the reference to x in the third line above would be ambiguous.

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, and
- 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.

(15) Assume the following two definitions of a objects named `X` in packages `P` and `Q`.

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

package Q {
  object X { val x = true; val y = "" }
}
```

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 A {
  println("L4: "+X)          // `X' refers to `P.X' here
  object B {
    import Q._               // `X' bound by wildcard import
    println("L7: "+X)        // `X' refers to `Q.X' here
    import X._               // `x' and `y' bound by wildcard import
    println("L8: "+x)        // `x' refers to `Q.X.x' here
    object C {
      val x = 3              // `x' bound by local definition
      println("L12: "+x)     // `x' refers to constant `3' here
      { import Q.X._         // `x' and `y' bound by wildcard import
//      println("L14: "+x)   // reference to `x' is ambiguous here
        import X.y           // `y' bound by explicit import
        println("L16: "+y)   // `y' refers to `Q.X.y' here
        { val x = "abc"      // `x' bound by local definition
          import P.X._       // `x' and `y' bound by wildcard import
//        println("L19: "+y) // reference to `y' is ambiguous here
          println("L20: "+x) // `x' refers to string ``abc'' here
}}}}}}
```

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.

Chapter 3

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’
                      |  ‘(’ 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.

[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).

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.

3.1 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 ε (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.

3.2 Value Types

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

3.2.1 Singleton Types

```
SimpleType  ::=  Path ‘.’ type
```

A singleton type is of the form `$p$.type`, where p is a path pointing to a value expected to conform to `scala.AnyRef`. The type denotes the set of values consisting of **`null`** and the value denoted by p.

A *stable type* is either a singleton type or a type which is declared to be a subtype of trait `scala.Singleton`.

3.2.2 Type Projection

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

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

3.2.3 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`.

(16) 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`,

t	□.type#t
Int	scala.type#Int
scala.Int	scala.type#Int
data.maintable.Node	data.maintable.type#Node

3.2.4 Parameterized Types

```
SimpleType      ::=  SimpleType TypeArgs
TypeArgs        ::=  ‘[’ Types ‘]’
```

A parameterized type $T[U_1, \ldots, U_n]$ consists of a type designator T and type parameters $U_1, \ldots, U_n$ 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 $[a_1 := T_1, \ldots, a_n := T_n]$.

(17) 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]
```

(18) Given the type definitions of (17), 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
```

3.2.5 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$n$[+T1, … , +$T_n$](_1: T1, … , _n: $T_n$)
extends Product_n[T1, … , $T_n$] {}

trait Product_n[+T1, … , +$T_n$] {
  override def arity = $n$
  def _1: T1
  …
  def _n: $T_n$
}
```

3.2.6 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.

(19) The following type adds the `@suspendable` annotation to the type `String`:

```
String @suspendable
```

3.2.7 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 function'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 }`.

(20) The following example shows how to declare and use a function which parameter's type 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 function `takeoff` is defined using a refinement with structural declarations to accept any object that declares a value `callsign` and a `fly` function.

[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.

3.2.8 Infix Types

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

An infix type `$T_1$ \mathit{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 `$\mathit{op}$[$T_1$, $T_2$]`. The infix operator *op* may be an arbitrary identifier, except for `*`, which is reserved as a postfix modifier denoting a repeated parameter type.

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 $t_0 \, \mathit{op} \, t_1 \, \mathit{op_2} \, \ldots \, \mathit{op_n} \, t_n$, all operators $\mathit{op}_1, \ldots, \mathit{op}_n$ must have the same associativity. If they are all left-associative, the sequence is interpreted as $(\ldots(t_0 \mathit{op_1} t_1) \mathit{op_2} \ldots) \mathit{op_n} t_n$, otherwise it is interpreted as $t_0 \mathit{op_1} (t_1 \mathit{op_2} (\ldots \mathit{op_n} t_n) \ldots)$.

3.2.9 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$_n$[T1 , … , $T_n$, U]`. Such class types are defined in the Scala library for n between 0 and 9 as follows.

```
package scala
trait Function_n[-T1 , … , -T$_n$, +R] {
  def apply(x1: T1 , … , x$_n$: T$_n$): R
  override def toString = "<function>"
}
```

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

3.2.10 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 $t_1[\mathit{tps}_1] >: L_1 <: U_1, \ldots, t_n[\mathit{tps}_n] >: L_n <: U_n$ be the types declared in Q (any of the type parameter sections `[ $\mathit{tps}_i$ ]` 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 $t[\mathit{tps}] >: L <: U$` 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 $T = p.c[\mathit{targs}, T, \mathit{targs}']$ be a parameterized type where $\mathit{targs}, \mathit{targs}'$ may be empty and T is a wildcard type `_$\;$>:$\,L\,$<:$\,U$`. Then T is equivalent to the existential type

```
$p.c[\mathit{targs},t,\mathit{targs}']$ 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.

(21) Assume the class definitions

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

Here are some examples of existential types:

```scala
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:

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

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

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

(23) 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]`.

3.3 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.

3.3.1 Method Types

A method type is denoted internally as $(\mathit{Ps})U$, 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: $(\mathit{Ps}_1)(\mathit{Ps}_2)U$ is treated as $(\mathit{Ps}_1)((\mathit{Ps}_2)U)$.

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.

(24) 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
```

3.3.2 Polymorphic Method Types

A polymorphic method type is denoted internally as `[$\mathit{tps}\,$]$T$` where `[$\mathit{tps}\,$]` 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.

(25) 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]   .
```

3.3.3 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.

(26) Consider this fragment of the `Iterable[+X]` class:

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

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`.

3.4 Base Types and Member Definitions

Types of class members depend on the way the members are referenced. Central here are three notions, namely: #. the notion of the set of base types of a type T, #. the notion of a type T in some class C seen from some prefix type S, #. 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 $\ldots$ with $T_n$ { $R$ }` are the *reduced union* of the base classes of all T_i's. This means: Let the multi-set $\mathscr{S}$ be the multi-set-union of the base types of all T_i's. If $\mathscr{S}$ 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 `$S$ = $\epsilon$.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, and (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. share a to

3.5 Relations between types

We define two relations between types.

Type equivalence	$T \equiv U$	T and U are interchangeable in all contexts.
Conformance	$T <: U$	Type T conforms to type U.

3.5.1 Type 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.

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

- Two method types are equivalent if they have equivalent result types, both 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.

3.5.2 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` and not `$T$ <: scala.NotNull` 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 $\ldots$ 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 $\ldots$ 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 $L <: t <: U$.

The $(<:)$ relation forms pre-order between types, i.e. it is transitive and reflexive. *least upper bounds* and *greatest lower bounds* of a set of types are understood to be relative to that order.

> **Note**: 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 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.

3.5.3 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  $<:_w$ Short
Short $<:_w$ Int
Char  $<:_w$ Int
Int   $<:_w$ Long
Long  $<:_w$ Float
Float $<:_w$ Double
```

[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.

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

3.6 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.

3.7 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 $\ldots$ 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}.

Chapter 4

Basic Declarations and Definitions

```
Dcl         ::=  ‘val’ ValDcl
              |  ‘var’ VarDcl
              |  ‘def’ FunDcl
              |  ‘type’ {nl} TypeDcl
PatVarDef   ::=  ‘val’ PatDef
              |  ‘var’ VarDef
Def         ::=  PatVarDef
              |  ‘def’ FunDef
              |  ‘type’ {nl} TypeDef
              |  TmplDef
```

A *declaration* introduces names and assigns them types. It can form part of a class definition or of a refinement in a compound type.

A *definition* introduces names that denote terms or types. It can form part of an object or class definition or it can be local to a block. Both declarations and definitions produce *bindings* that associate type names with type definitions or bounds, and that associate term names with types.

The scope of a name introduced by a declaration or definition is the whole statement sequence containing the binding. However, there is a restriction on forward references in blocks: In a statement sequence $s_1 \ldots s_n$ making up a block, if a simple name in s_i refers to an entity defined by s_j where $j \geq i$, then for all s_k between and including s_i and s_j,

- s_k cannot be a variable definition.
- If s_k is a value definition, it must be lazy.

4.1 Value Declarations and Definitions

```
Dcl           ::=  ‘val’ ValDcl
ValDcl        ::=  ids ‘:’ Type
PatVarDef     ::=  ‘val’ PatDef
PatDef        ::=  Pattern2 {‘,’ Pattern2} [‘:’ Type] ‘=’ Expr
ids           ::=  id {‘,’ id}
```

A value declaration `val $x$: $T$` introduces x as a name of a value of type T.

A value definition `val $x$: $T$ = $e$` defines x as a name of the value that results from the evaluation of e. If the value definition is not recursive, the type T may be omitted, in which case the packed type of expression e is assumed. If a type T is given, then e is expected to conform to it.

Evaluation of the value definition implies evaluation of its right-hand side e, unless it has the modifier `lazy`. The effect of the value definition is to bind x to the value of e converted to type T. A `lazy` value definition evaluates its right hand side e the first time the value is accessed.

A *constant value definition* is of the form

```
final val x = e
```

where `e` is a constant expression. The `final` modifier must be present and no type annotation may be given. References to the constant value `x` are themselves treated as constant expressions; in the generated code they are replaced by the definition's right-hand side `e`.

Value definitions can alternatively have a pattern as left-hand side. If p is some pattern other than a simple name or a name followed by a colon and a type, then the value definition `val $p$ = $e$` is expanded as follows:

1. If the pattern p has bound variables $x_1, \ldots, x_n$, where $n > 1$:

```
val $\$ x$ = $e$ match {case $p$ => {$x_1 , \ldots , x_n$}}
val $x_1$ = $\$ x$._1
$\ldots$
val $x_n$ = $\$ x$._n  .
```

Here, $\$x$ is a fresh name.

2. If p has a unique bound variable x:

```
val $x$ = $e$ match { case $p$ => $x$ }
```

3. If p has no bound variables:

```
$e$ match { case $p$ => ()}
```

(27) The following are examples of value definitions

```
val pi = 3.1415
val pi: Double = 3.1415   // equivalent to first definition
val Some(x) = f()         // a pattern definition
val x :: xs = mylist      // an infix pattern definition
```

The last two definitions have the following expansions.

```
val x = f() match { case Some(x) => x }

val x$\$$ = mylist match { case x :: xs => {x, xs} }
val x = x$\$$._1
val xs = x$\$$._2
```

The name of any declared or defined value may not end in `_=`.

A value declaration `val $x_1 , \ldots , x_n$: $T$` is a shorthand for the sequence of value declarations `val $x_1$: $T$; ...; val $x_n$: $T$`. A value definition `val $p_1 , \ldots , p_n$ = $e$` is a shorthand for the sequence of value definitions `val $p_1$ = $e$; ...; val $p_n$ = $e$`. A value definition `val $p_1 , \ldots , p_n: T$ = $e$` is a shorthand for the sequence of value definitions `val $p_1: T$ = $e$; ...; val $p_n: T$ = $e$`.

4.2 Variable Declarations and Definitions

```
Dcl           ::=  ‘var’ VarDcl
PatVarDef     ::=  ‘var’ VarDef
VarDcl        ::=  ids ‘:’ Type
VarDef        ::=  PatDef
                |  ids ‘:’ Type ‘=’ ‘_’
```

A variable declaration `var $x$: $T$` is equivalent to declarations of a *getter function* x and a *setter function* `$x$_=`, defined as follows:

```
def $x$: $T$
def $x$_= ($y$: $T$): Unit
```

An implementation of a class containing variable declarations may define these variables using variable definitions, or it may define setter and getter functions directly.

A variable definition `var $x$: $T$ = $e$` introduces a mutable variable with type T and initial value as given by the expression e. The type T can be omitted, in which case the type of e is assumed. If T is given, then e is expected to conform to it.

Variable definitions can alternatively have a pattern as left-hand side. A variable definition `var $p$ = $e$` where p is a pattern other than a simple name or a name followed by a colon and a type is expanded in the same way as a value definition `val $p$ = $e$`, except that the free names in p are introduced as mutable variables, not values.

The name of any declared or defined variable may not end in `_=`.

A variable definition `var $x$: $T$ = _` can appear only as a member of a template. It introduces a mutable field with type T and a default initial value. The default value depends on the type T as follows:

`0`	if T is `Int` or one of its subrange types
`0L`	if T is `Long`
`0.0f`	if T is `Float`
`0.0d`	if T is `Double`
`false`	if T is `Boolean`
`{}`	if T is `Unit`
`null`	for all other types T

When they occur as members of a template, both forms of variable definition also introduce a getter function x which returns the value currently assigned to the variable, as well as a setter function `$x$_=` which changes the value currently assigned to the variable. The functions have the same signatures as for a variable declaration. The template then has these getter and setter functions as members, whereas the original variable cannot be accessed directly as a template member.

(28) The following example shows how *properties* can be simulated in Scala. It defines a class `TimeOfDayVar` of time values with updatable integer fields representing hours, minutes, and seconds. Its implementation contains tests that allow only legal values to be assigned to these fields. The user code, on the other hand, accesses these fields just like normal variables.

```
class TimeOfDayVar {
  private var h: Int = 0
  private var m: Int = 0
  private var s: Int = 0
```

```
    def hours              =  h
    def hours_= (h: Int)   =  if (0 <= h && h < 24) this.h = h
                              else throw new DateError()

    def minutes            =  m
    def minutes_= (m: Int) =  if (0 <= m && m < 60) this.m = m
                              else throw new DateError()

    def seconds            =  s
    def seconds_= (s: Int) =  if (0 <= s && s < 60) this.s = s
                              else throw new DateError()
  }
  val d = new TimeOfDayVar
  d.hours = 8; d.minutes = 30; d.seconds = 0
  d.hours = 25                  // throws a DateError exception
```

A variable declaration `var $x_1 , \ldots , x_n$: $T$` is a shorthand for the sequence of variable declarations `var $x_1$: $T$; ...; var $x_n$: $T$`. A variable definition `var $x_1 , \ldots , x_n$ = $e$` is a shorthand for the sequence of variable definitions `var $x_1$ = $e$; ...; var $x_n$ = $e$`. A variable definition `var $x_1 , \ldots , x_n: T$ = $e$` is a shorthand for the sequence of variable definitions `var $x_1: T$ = $e$; ...; var $x_n: T$ = $e$`.

4.3 Type Declarations and Type Aliases

```
Dcl        ::=  ‘type’ {nl} TypeDcl
TypeDcl    ::=  id [TypeParamClause] [‘>:’ Type] [‘<:’ Type]
Def        ::=  type {nl} TypeDef
TypeDef    ::=  id [TypeParamClause] ‘=’ Type
```

A *type declaration* `type $t$[$\mathit{tps}\,$] >: $L$ <: $U$` declares t to be an abstract type with lower bound type L and upper bound type U. If the type parameter clause `[$\mathit{tps}\,$]` is omitted, t abstracts over a first-order type, otherwise t stands for a type constructor that accepts type arguments as described by the type parameter clause.

If a type declaration appears as a member declaration of a type, implementations of the type may implement t with any type T for which $L <: T <: U$. It is a compile-time error if L does not conform to U. Either or both bounds may be omitted. If the lower bound L is absent, the bottom type `scala.Nothing` is assumed. If the upper bound U is absent, the top type `scala.Any` is assumed.

A type constructor declaration imposes additional restrictions on the concrete types for which t may stand. Besides the bounds L and U, the type parameter clause may impose higher-order bounds and variances, as governed by the conformance of type constructors.

The scope of a type parameter extends over the bounds `>: $L$ <: $U$` and the type parameter clause *tps* itself. A higher-order type parameter clause (of an abstract type constructor tc) has the same kind of scope, restricted to the declaration of the type parameter tc.

To illustrate nested scoping, these declarations are all equivalent: `type t[m[x] <: Bound[x], Bound[x]]`, `type t[m[x] <: Bound[x], Bound[y]]` and `type t[m[x] <: Bound[x], Bound[_]]`, as the scope of, e.g., the type parameter of m is limited to the declaration of m. In all of them, t is an abstract type member that abstracts over two type constructors: m stands for a type constructor that takes one type parameter and that must be a subtype of $Bound$, t's second type constructor parameter. `t[MutableList, Iterable]` is a valid use of t.

A *type alias* `type $t$ = $T$` defines t to be an alias name for the type T. The left hand side of a type alias may have a type parameter clause, e.g. `type $t$[$\mathit{tps}\,$] = $T$`. The scope of a type parameter extends over the right hand side T and the type parameter clause *tps* itself.

The scope rules for definitions and type parameters make it possible that a type name appears in its own bound or in its right-hand side. However, it is a static error if a type alias refers recursively to the defined type constructor itself.
That is, the type T in a type alias `type $t$[$\mathit{tps}\,$] = $T$` may not refer directly or indirectly to the name t. It is also an error if an abstract type is directly or indirectly its own upper or lower bound.

(29) The following are legal type declarations and definitions:

```
type IntList = List[Integer]
type T <: Comparable[T]
type Two[A] = Tuple2[A, A]
type MyCollection[+X] <: Iterable[X]
```

The following are illegal:

```
type Abs = Comparable[Abs]      // recursive type alias

type S <: T                     // S, T are bounded by themselves.
type T <: S

type T >: Comparable[T.That]    // Cannot select from T.
                                // T is a type, not a value
type MyCollection <: Iterable   // Type constructor members must explicitly
                                // state their type parameters.
```

If a type alias `type $t$[$\mathit{tps}\,$] = $S$` refers to a class type S, the name t can also be used as a constructor for objects of type S.

(30) The Predef object contains a definition which establishes Pair as an alias of the parameterized class Tuple2:

```
type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

As a consequence, for any two types S and T, the type `Pair[$S$, $T\,$]` is equivalent to the type `Tuple2[$S$, $T\,$]`. Pair can also be used as a constructor instead of Tuple2, as in:

```
val x: Pair[Int, String] = new Pair(1, "abc")
```

4.4 Type Parameters

```
TypeParamClause  ::= '[' VariantTypeParam {',' VariantTypeParam} ']'
VariantTypeParam ::= {Annotation} ['+' | '-'] TypeParam
TypeParam        ::= (id | '_') [TypeParamClause] ['>:' Type] ['<:' Type] [':' Type]
```

Type parameters appear in type definitions, class definitions, and function definitions. In this section we consider only type parameter definitions with lower bounds `>: $L$` and upper bounds `<: $U$` whereas a discussion of context bounds `: $U$` and view bounds `<% $U$` is deferred to here.

The most general form of a first-order type parameter is `$@a_1 \ldots @a_n$ $\pm$ $t$ >: $L$ <: $U$`. Here, L, and U are lower and upper bounds that constrain possible type arguments for the parameter. It is a compile-time error if L does not conform to U. $\pm$ is a *variance*, i.e. an optional prefix of either +, or -. One or more annotations may precede the type parameter.

The names of all type parameters must be pairwise different in their enclosing type parameter clause. The scope of a type parameter includes in each case the whole type parameter clause. Therefore it is possible that a type parameter appears as part of its own bounds or the bounds of other type parameters in the same clause. However, a type parameter may not be bounded directly or indirectly by itself.
A type constructor parameter adds a nested type parameter clause to the type parameter. The most general form of a type constructor parameter is `$@a_1\ldots@a_n$ $\pm$ $t[\mathit{tps}\,]$ >: $L$ <: $U$`.

The above scoping restrictions are generalized to the case of nested type parameter clauses, which declare higher-order type parameters. Higher-order type parameters (the type parameters of a type parameter t) are only visible in their immediately surrounding parameter clause (possibly including clauses at a deeper nesting level) and in the bounds of t. Therefore, their names must only be pairwise different from the names of other visible parameters. Since the names of higher-order type parameters are thus often irrelevant, they may be denoted with a '_', which is nowhere visible.

(31) Here are some well-formed type parameter clauses:

```
[S, T]
[@specialized T, U]
[Ex <: Throwable]
[A <: Comparable[B], B <: A]
[A, B >: A, C >: A <: B]
[M[X], N[X]]
[M[_], N[_]] // equivalent to previous clause
[M[X <: Bound[X]], Bound[_]]
[M[+X] <: Iterable[X]]
```

The following type parameter clauses are illegal:

```
[A >: A]                  // illegal, 'A' has itself as bound
[A <: B, B <: C, C <: A]  // illegal, 'A' has itself as bound
[A, B, C >: A <: B]       // illegal lower bound 'A' of 'C' does
                          // not conform to upper bound 'B'.
```

4.5 Variance Annotations

Variance annotations indicate how instances of parameterized types vary with respect to subtyping. A '+' variance indicates a covariant dependency, a '-' variance indicates a contravariant dependency, and a missing variance indication indicates an invariant dependency.

A variance annotation constrains the way the annotated type variable may appear in the type or class which binds the type parameter. In a type definition `type $T$[$\mathit{tps}\,$] = $S$`, or a type declaration `type $T$[$\mathit{tps}\,$] >: $L$ <: $U$` type parameters labeled '+' must only appear in covariant position whereas type parameters labeled '-' must only appear in contravariant position. Analogously, for a class definition `class $C$[$\mathit{tps}\,$]($\mathit{ps}\,$) extends $T$ { $x$: $S$ => ...}`, type parameters labeled '+' must only appear in covariant position in the self type S and the template T, whereas type parameters labeled '-' must only appear in contravariant position.

The variance position of a type parameter in a type or template is defined as follows. Let the opposite of covariance be contravariance, and the opposite of invariance be itself. The top-level of the type or template is always in covariant position. The variance position changes at the following constructs.

- The variance position of a method parameter is the opposite of the variance position of the enclosing parameter clause.

- The variance position of a type parameter is the opposite of the variance position of the enclosing type parameter clause.
- The variance position of the lower bound of a type declaration or type parameter is the opposite of the variance position of the type declaration or parameter.

- The type of a mutable variable is always in invariant position.
- The prefix S of a type selection `$S$#$T$` is always in invariant position.
- For a type argument T of a type `$S$[$\ldots T \ldots$ ]`: If the corresponding type parameter is invariant, then T is in invariant position. If the corresponding type parameter is contravariant, the variance position of T is the opposite of the variance position of the enclosing type `$S$[$\ldots T \ldots$ ]`.

References to the type parameters in object-private values, variables, or methods of the class are not checked for their variance position. In these members the type parameter may appear anywhere without restricting its legal variance annotations.

(32) The following variance annotation is legal.

```
abstract class P[+A, +B] {
  def fst: A; def snd: B
}
```

With this variance annotation, type instances of P subtype covariantly with respect to their arguments. For instance,

```
P[IOException, String] <: P[Throwable, AnyRef]
```

If the members of P are mutable variables, the same variance annotation becomes illegal.

```
abstract class Q[+A, +B](x: A, y: B) {
  var fst: A = x          // **** error: illegal variance:
  var snd: B = y          // ‘A’, ‘B’ occur in invariant position.
}
```

If the mutable variables are object-private, the class definition becomes legal again:

```
abstract class R[+A, +B](x: A, y: B) {
  private[this] var fst: A = x        // OK
  private[this] var snd: B = y        // OK
}
```

(33) The following variance annotation is illegal, since a appears in contravariant position in the parameter of append:

```
abstract class Sequence[+A] {
  def append(x: Sequence[A]): Sequence[A]
                  // **** error: illegal variance:
                  // ‘A’ occurs in contravariant position.
}
```

The problem can be avoided by generalizing the type of `append` by means of a lower bound:

```
abstract class Sequence[+A] {
  def append[B >: A](x: Sequence[B]): Sequence[B]
}
```

(34) Here is a case where a contravariant type parameter is useful.

```
abstract class OutputChannel[-A] {
  def write(x: A): Unit
}
```

With that annotation, we have that `OutputChannel[AnyRef]` conforms to `OutputChannel[String]`.
That is, a channel on which one can write any object can substitute for a channel on which one can write only strings.

4.6 Function Declarations and Definitions

```
Dcl                ::=  ‘def’ FunDcl
FunDcl             ::=  FunSig ‘:’ Type
Def                ::=  ‘def’ FunDef
FunDef             ::=  FunSig [‘:’ Type] ‘=’ Expr
FunSig             ::=  id [FunTypeParamClause] ParamClauses
FunTypeParamClause ::=  ‘[’ TypeParam {‘,’ TypeParam} ‘]’
ParamClauses       ::=  {ParamClause} [[nl] ‘(’ ‘implicit’ Params ‘)’]
ParamClause        ::=  [nl] ‘(’ [Params] ‘)’}
Params             ::=  Param {‘,’ Param}
Param              ::=  {Annotation} id [‘:’ ParamType] [‘=’ Expr]
ParamType          ::=  Type
                     |  ‘=>’ Type
                     |  Type ‘*’
```

A function declaration has the form `def $f\,\mathit{psig}$: $T$`, where f is the function's name, *psig* is its parameter signature and T is its result type. A function definition `def $f\,\mathit{psig}$: $T$ = $e$` also includes a *function body* e, i.e. an expression which defines the function's result. A parameter signature consists of an optional type parameter clause `[$\mathit{tps}\,$]`, followed by zero or more value parameter clauses `($\mathit{ps}_1$)$\ldots$($\mathit{ps}_n$)`. Such a declaration or definition introduces a value with a (possibly polymorphic) method type whose parameter types and result type are as given.

The type of the function body is expected to conform to the function's declared result type, if one is given. If the function definition is not recursive, the result type may be omitted, in which case it is determined from the packed type of the function body.

A type parameter clause *tps* consists of one or more type declarations, which introduce type parameters, possibly with bounds. The scope of a type parameter includes the whole signature, including any of the type parameter bounds as well as the function body, if it is present.

A value parameter clause *ps* consists of zero or more formal parameter bindings such as `$x$: $T$` or `$x: T = e$`, which bind value parameters and associate them with their types. Each value parameter declaration may optionally define a default argument. The default argument expression e is type-checked with an expected type T' obtained by replacing all occurences of the function's type parameters in T by the undefined type.

For every parameter $p_{i,j}$ with a default argument a method named `$f\$$default$\$$n` is generated which computes the default argument expression. Here, n denotes the parameter's position in the method declaration. These methods are parametrized by the type parameter clause `[$\mathit{tps}\,$]` and all value parameter clauses `($\mathit{ps}_1$)$\ldots$($\mathit{ps}_{i-1}$)` preceeding $p_{i,j}$. The `$f\$$default$\$$n` methods are inaccessible for user programs.

The scope of a formal value parameter name x comprises all subsequent parameter clauses, as well as the method return type and the function body, if they are given.[1] Both type parameter names and value parameter names must be pairwise distinct.

(35) In the method

```
def compare[T](a: T = 0)(b: T = a) = (a == b)
```

the default expression `0` is type-checked with an undefined expected type. When applying `compare()`, the default value `0` is inserted and `T` is instantiated to `Int`. The methods computing the default arguments have the form:

```
def compare$\$$default$\$$1[T]: Int = 0
def compare$\$$default$\$$2[T](a: T): T = a
```

4.6.1 By-Name Parameters

```
ParamType          ::=  '=>' Type
```

[1] However, at present singleton types of method parameters may only appear in the method body; so *dependent method types* are not supported.

The type of a value parameter may be prefixed by `=>`, e.g. `$x$: => $T$`. The type of such a parameter is then the parameterless method type `=> $T$`. This indicates that the corresponding argument is not evaluated at the point of function application, but instead is evaluated at each use within the function. That is, the argument is evaluated using *call-by-name*.

The by-name modifier is disallowed for parameters of classes that carry a `val` or `var` prefix, including parameters of case classes for which a `val` prefix is implicitly generated. The by-name modifier is also disallowed for implicit parameters.

(36) The declaration

```
def whileLoop (cond: => Boolean) (stat: => Unit): Unit
```

indicates that both parameters of `whileLoop` are evaluated using call-by-name.

4.6.2 Repeated Parameters

```
ParamType          ::=  Type ‘*’
```

The last value parameter of a parameter section may be suffixed by “*”, e.g. `(..., $x$:$T$*)`. The type of such a *repeated* parameter inside the method is then the sequence type `scala.Seq[$T$]`. Methods with repeated parameters `$T$*` take a variable number of arguments of type T. That is, if a method m with type `($p_1:T_1 , \ldots , p_n:T_n, p_s:S$*)$U$` is applied to arguments $(e_1, \ldots, e_k)$ where $k \geq n$, then m is taken in that application to have type $(p_1 : T_1, \ldots, p_n : T_n, p_s : S, \ldots, p_{s'} S)U$, with $k - n$ occurrences of type S where any parameter names beyond p_s are fresh. The only exception to this rule is if the last argument is marked to be a *sequence argument* via a `_*` type annotation. If m above is applied to arguments `($e_1 , \ldots , e_n, e'$: _*)`, then the type of m in that application is taken to be `($p_1:T_1, \ldots , p_n:T_n,p_{s}:$scala.Seq[$S$])`.

It is not allowed to define any default arguments in a parameter section with a repeated parameter.

(37) The following method definition computes the sum of the squares of a variable number of integer arguments.

```
def sum(args: Int*) = {
  var result = 0
  for (arg <- args) result += arg * arg
  result
}
```

The following applications of this method yield `0`, `1`, `6`, in that order.

```
sum()
sum(1)
sum(1, 2, 3)
```

Furthermore, assume the definition:

```
val xs = List(1, 2, 3)
```

The following application of method `sum` is ill-formed:

```
sum(xs)       // ***** error: expected: Int, found: List[Int]
```

By contrast, the following application is well formed and yields again the result 6:

```
sum(xs: _*)
```

4.6.3 Procedures

```
FunDcl   ::=  FunSig
FunDef   ::=  FunSig [nl] '{' Block '}'
```

Special syntax exists for procedures, i.e. functions that return the `Unit` value `{}`. A procedure declaration is a function declaration where the result type is omitted. The result type is then implicitly completed to the `Unit` type. E.g., `def $f$($\mathit{ps}$)` is equivalent to `def $f$($\mathit{ps}$): Unit`.

A procedure definition is a function definition where the result type and the equals sign are omitted; its defining expression must be a block. E.g., `def $f$($\mathit{ps}$) {$\mathit{stats}$}` is equivalent to `def $f$($\mathit{ps}$): Unit = {$\mathit{stats}$}`.

(38) Here is a declaration and a definition of a procedure named `write`:

```
trait Writer {
  def write(str: String)
}
object Terminal extends Writer {
  def write(str: String) { System.out.println(str) }
}
```

The code above is implicitly completed to the following code:

```
trait Writer {
  def write(str: String): Unit
}
object Terminal extends Writer {
  def write(str: String): Unit = { System.out.println(str) }
}
```

4.6.4 Method Return Type Inference

A class member definition m that overrides some other function m' in a base class of C may leave out the return type, even if it is recursive. In this case, the return type R' of the overridden function m', seen as a member of C, is taken as the return type of m for each recursive invocation of m. That way, a type R for the right-hand side of m can be determined, which is then taken as the return type of m. Note that R may be different from R', as long as R conforms to R'.

(39) Assume the following definitions:

```
trait I {
  def factorial(x: Int): Int
}
class C extends I {
  def factorial(x: Int) = if (x == 0) 1 else x * factorial(x - 1)
}
```

Here, it is OK to leave out the result type of `factorial` in `C`, even though the method is recursive.

4.7 Import Clauses

```
Import          ::= ‘import’ ImportExpr {‘,’ ImportExpr}
ImportExpr      ::= StableId ‘.’ (id | ‘_’ | ImportSelectors)
ImportSelectors ::= ‘{’ {ImportSelector ‘,’}
                    (ImportSelector | ‘_’) ‘}’
ImportSelector  ::= id [‘=>’ id | ‘=>’ ‘_’]
```

An import clause has the form `import $p$.$I$` where p is a stable identifier and I is an import expression. The import expression determines a set of names of importable members of p which are made available without qualification. A member m of p is *importable* if it is not object-private. The most general form of an import expression is a list of {*import selectors}*

`{ $x_1$ => $y_1 , \ldots , x_n$ => $y_n$, _ }`

for $n \geq 0$, where the final wildcard ‘_’ may be absent. It makes available each importable member `$p$.$x_i$` under the unqualified name y_i. I.e. every import selector `$x_i$ => $y_i$` renames `$p$.$x_i$` to y_i. If a final wildcard is present, all importable members z of p other than `$x_1 , \ldots , x_n,y_1 , \ldots , y_n$` are also made available under their own unqualified names.

Import selectors work in the same way for type and term members. For instance, an import clause `import $p$.{$x$ => $y\,$}` renames the term name `$p$.$x$` to the term name

y and the type name p.x to the type name y. At least one of these two names must reference an importable member of p.

If the target in an import selector is a wildcard, the import selector hides access to the source member. For instance, the import selector `$x$ => _` "renames" x to the wildcard symbol (which is unaccessible as a name in user programs), and thereby effectively prevents unqualified access to x. This is useful if there is a final wildcard in the same import selector list, which imports all members not mentioned in previous import selectors.

The scope of a binding introduced by an import-clause starts immediately after the import clause and extends to the end of the enclosing block, template, package clause, or compilation unit, whichever comes first.

Several shorthands exist. An import selector may be just a simple name x. In this case, x is imported without renaming, so the import selector is equivalent to `$x$ => $x$`. Furthermore, it is possible to replace the whole import selector list by a single identifier or wildcard. The import clause `import $p$.$x$` is equivalent to `import $p$.{$x\,$}`, i.e. it makes available without qualification the member x of p. The import clause `import $p$._` is equivalent to `import $p$.{_}`, i.e. it makes available without qualification all members of p (this is analogous to `import $p$.*` in Java).

An import clause with multiple import expressions `import $p_1$.$I_1 , \ldots , p_n$.$I_n$` is interpreted as a sequence of import clauses `import $p_1$.$I_1$; $\ldots$; import $p_n$.$I_n$`.

(40) *Consider the object definition:*

```
object M {
  def z = 0, one = 1
  def add(x: Int, y: Int): Int = x + y
}
```

Then the block

```
{ import M.{one, z => zero, _}; add(zero, one) }
```

is equivalent to the block

```
{ M.add(M.z, M.one) }
```

Chapter 5

Classes and Objects

```
TmplDef          ::= ['case'] 'class' ClassDef
                  |  ['case'] 'object' ObjectDef
                  |  'trait' TraitDef
```

Classes and objects are both defined in terms of templates.

5.1 Templates

```
ClassTemplate   ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate   ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents    ::=  Constr {'with' AnnotType}
TraitParents    ::=  AnnotType {'with' AnnotType}
TemplateBody   ::= [nl] '{' [SelfType] TemplateStat {semi TemplateStat} '}'
SelfType        ::=  id [':' Type] '=>'
                 |   this ':' Type '=>'
```

A template defines the type signature, behavior and initial state of a trait or class of objects or of a single object. Templates form part of instance creation expressions, class definitions, and object definitions. A template `$sc$ with $mt_1$ with $\ldots$ with $mt_n$ { $\mathit{stats}$ }` consists of a constructor invocation sc *which defines the template's* superclass*, trait references `$mt_1 , \ldots , mt_n$`* $(n \geq 0)$*, which define the template's* traits*, and a statement sequence stats which contains initialization code and additional member definitions for the template.*

Each trait reference mt_i *must denote a trait. By contrast, the superclass constructor* sc *normally refers to a class which is not a trait. It is possible to write a list of parents that starts with a trait reference, e.g. `$mt_1$ with $\ldots$ with $mt_n$`. In that case the list of parents is implicitly extended to include the supertype of* mt_1 *as first parent type. The new* *supertype must have at least one constructor that does not take parameters. In the following, we will always assume that this implicit extension has been performed, so that the first parent class of a template is a regular superclass constructor, not a trait reference.*

The list of parents of every class is also always implicitly extended by a reference to the `scala.ScalaObject` *trait as last mixin. E.g.*

```
$sc$ with $mt_1$ with $\ldots$ with $mt_n$ { $\mathit{stats}$ }
```

becomes

```
$mt_1$ with $\ldots$ with $mt_n$ with ScalaObject { $\mathit{stats}$ }.
```

The list of parents of a template must be well-formed. This means that the class denoted by the superclass constructor sc *must be a subclass of the superclasses of all the traits* $mt_1, \ldots, mt_n$. *In other words, the non-trait classes inherited by a template form a chain in the inheritance hierarchy which starts with the template's superclass.*

The least proper supertype *of a template is the class type or compound type consisting of all its parent class types.*

The statement sequence stats contains member definitions that define new members or overwrite members in the parent classes. If the template forms part of an abstract class or trait definition, the statement part stats may also contain declarations of abstract members. If the template forms part of a concrete class definition, stats may still contain declarations of abstract type members, but not of abstract term members. Furthermore, stats may in any case also contain expressions; these are executed in the order they are given as part of the initialization of a template.

The sequence of template statements may be prefixed with a formal parameter definition and an arrow, e.g. `$x$ =>`*, or* `$x$:$T$ =>`*. If a formal parameter is given, it can be used as an alias for the reference* `this` *throughout the body of the template.*
If the formal parameter comes with a type T*, this definition affects the* self type S *of the underlying class or object as follows: Let* C *be the type of the class or trait or object defining the template. If a type* T *is given for the formal self parameter,* S *is the greatest lower bound of* T *and* C*. If no type* T *is given,* S *is just* C*. Inside the template, the type of* `this` *is assumed to be* S*.*

The self type of a class or object must conform to the self types of all classes which are inherited by the template t*.*

A second form of self type annotation reads just `this: $S$ =>`*. It prescribes the type* S *for* `this` *without introducing an alias name for it.*

(41) *Consider the following class definitions:*

```
class Base extends Object {}
trait Mixin extends Base {}
object O extends Mixin {}
```

In this case, the definition of O is expanded to:

```
object O extends Base with Mixin {}
```

Inheriting from Java Types
A template may have a Java class as its superclass and Java interfaces as its mixins.

Template Evaluation
Consider a template `$sc$ with $mt_1$ with $mt_n$ { $\mathit{stats}$ }`.

If this is the template of a trait then its mixin-evaluation *consists of an evaluation of the statement sequence stats.*

If this is not a template of a trait, then its evaluation *consists of the following steps.*

- *First, the superclass constructor sc is evaluated.*
- *Then, all base classes in the template's linearization up to the template's superclass denoted by sc are mixin-evaluated. Mixin-evaluation happens in reverse order of occurrence in the linearization.*
- *Finally the statement sequence stats is evaluated.*

Delayed Initializaton
The initialization code of an object or class (but not a trait) that follows the superclass constructor invocation and the mixin-evaluation of the template's base classes is passed to a special hook, which is inaccessible from user code. Normally, that hook simply executes the code that is passed to it. But templates inheriting the `scala.DelayedInit` trait can override the hook by re-implementing the `delayedInit` method, which is defined as follows:

```
def delayedInit(body: => Unit)
```

5.1.1 Constructor Invocations

```
Constr  ::=  AnnotType {‘(’ [Exprs] ‘)’}
```

Constructor invocations define the type, members, and initial state of objects created by an instance creation expression, or of parts of an object's definition which are inherited by a class or object definition. A constructor invocation is a function application `$x$.$c$[$\mathit{targs}$]($\mathit{args}_1$)$\ldots$($\mathit{args}_n$)`, where x is a stable identifier, c is a type name which either designates a class or defines an alias type for one, targs is a type argument list, $\mathit{args}_1, \ldots, \mathit{args}_n$ are argument lists, and there is a constructor of that class which is applicable to the given arguments. If the constructor invocation uses named or default arguments, it is transformed into a block expression using the same transformation as described here.

The prefix '`$x$.`' can be omitted. A type argument list can be given only if the class c takes type parameters. Even then it can be omitted, in which case a type argument list is synthesized using local type inference. If no explicit arguments are given, an empty list `()` is implicitly supplied.

An evaluation of a constructor invocation `$x$.$c$[$\mathit{targs}$]($\mathit{args}_1$)$\ldots$($\mat` consists of the following steps:

- *First, the prefix x is evaluated.*
- *Then, the arguments $args_1, \ldots, args_n$ are evaluated from left to right.*
- *Finally, the class being constructed is initialized by evaluating the template of the class referred to by c.*

5.1.2 Class Linearization

The classes reachable through transitive closure of the direct inheritance relation from a class C are called the base classes *of C. Because of mixins, the inheritance relationship on base classes forms in general a directed acyclic graph. A linearization of this graph is defined as follows.*

> *Let C be a class with template `$C_1$ with ... with $C_n$ { $\mathit{stats}$ }`. The* linearization *of C, $\mathcal{L}(C)$ is defined as follows:*
>
> $\mathcal{L}(C) = C, \mathcal{L}(C_n) \vec{+} \ldots \vec{+} \mathcal{L}(C_1)$
>
> *Here $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left operand:*

```
\[
\begin{array}{lcll}
\{a, A\} \;\vec{+}\; B &=& a, (A \;\vec{+}\; B)  &{\bf if} \; a \not\in B \\
                       &=& A \;\vec{+}\; B       &{\bf if} \; a \in B
\end{array}
\]
```

(42) *Consider the following class definitions.*

```
abstract class AbsIterator extends AnyRef { ... }
trait RichIterator extends AbsIterator { ... }
class StringIterator extends AbsIterator { ... }
class Iter extends StringIterator with RichIterator { ... }
```

Then the linearization of class `Iter` is

```
{ Iter, RichIterator, StringIterator, AbsIterator, ScalaObject, AnyRef, Any }
```

Trait `ScalaObject` appears in this list because it is added as last mixin to every Scala class (see here).

Note that the linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur. **??** *also satisfies the property that a linearization of a class always contains the linearization of its direct superclass as a suffix. For instance, the linearization of `StringIterator` is*

```
{ StringIterator, AbsIterator, ScalaObject, AnyRef, Any }
```

which is a suffix of the linearization of its subclass `Iter`. The same is not true for the linearization of mixins. For instance, the linearization of `RichIterator` is

```
{ RichIterator, AbsIterator, ScalaObject, AnyRef, Any }
```

which is not a suffix of the linearization of `Iter`.

5.1.3 Class Members

A class C defined by a template `$C_1$ with $\ldots$ with $C_n$ { $\mathit{stats}$ }` can define members in its statement sequence stats and can inherit members from all parent classes. Scala adopts Java and C#'s conventions for static overloading of methods. It is thus possible that a class defines and/or inherits several methods with the same name. To decide whether a defined member of a class C overrides a member of a parent class, or whether the two co-exist as overloaded variants in C, Scala uses the following definition of matching *on members:*

Definition
A member definition M matches *a member definition M', if M and M' bind the same name, and one of following holds.*

1. *Neither M nor M' is a method definition.*
2. *M and M' define both monomorphic methods with equivalent argument types.*
3. *M defines a parameterless method and M' defines a method with an empty parameter list `()` or* vice versa.
4. *M and M' define both polymorphic methods with equal number of argument types $\overline{T}$, $\overline{T}'$ and equal numbers of type parameters $\overline{t}$, $\overline{t}'$, say, and $\overline{T}' = [\overline{t}'/\overline{t}]\overline{T}$. by the corresponding type parameter t of M.*

Member definitions fall into two categories: concrete and abstract. Members of class C are either directly defined *(i.e. they appear in C's statement sequence stats) or they are* inherited. *There are two rules that determine the set of members of a class, one for each category:*

> *A* concrete member *of a class C is any concrete definition M in some class $C_i \in \mathcal{L}(C)$, except if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which directly defines a concrete member M' matching M.*
>
> *An* abstract member *of a class C is any abstract definition M in some class $C_i \in \mathcal{L}(C)$, except if C contains already a concrete member M' matching M, or if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which directly defines an abstract member M' matching M.*

This definition also determines the overriding relationships between matching members of a class C and its parents.
First, a concrete definition always overrides an abstract definition. Second, for definitions M and M' which are both concrete or both abstract, M overrides M' if M appears in a class that precedes (in the linearization of C) the class in which M' is defined.

It is an error if a template directly defines two matching members. It is also an error if a template contains two members (directly defined or inherited) with the same name and the same erased type. Finally, a template is not allowed to contain two methods (directly defined or inherited) with the same name which both define default arguments.

(43) *Consider the trait definitions:*

```
trait A { def f: Int }
trait B extends A { def f: Int = 1 ; def g: Int = 2 ; def h: Int = 3 }
trait C extends A { override def f: Int = 4 ; def g: Int }
trait D extends B with C { def h: Int }
```

Then trait `D` has a directly defined abstract member `h`. It inherits member `f` from trait `C` and member `g` from trait `B`.

5.1.4 Overriding

A member M of class C that matches a non-private member M' of a base class of C is said to override *that member. In this case the binding of the overriding member M must subsume the binding of the overridden member M'. Furthermore, the following restrictions on modifiers apply to M and M':*

- *M' must not be labeled `final`.*
- *M must not be `private`.*
- *If M is labeled `private[$C$]` for some enclosing class or package C, then M' must be labeled `private[$C'$]` for some class or package C' where C' equals C or C' is contained in C.*
- *If M is labeled `protected`, then M' must also be labeled `protected`.*
- *If M' is not an abstract member, then M must be labeled `override`. Furthermore, one of two possibilities must hold:*

  - *either M is defined in a subclass of the class where is M' is defined,*
  - *or both M and M' override a third member M'' which is defined in a base class of both the classes containing M and M'*

- *If M' is incomplete in C then M must be labeled* `abstract override`.
- *If M and M' are both concrete value definitions, then either none of them is marked* `lazy` *or both must be marked* `lazy`.

A special rule concerns parameterless methods. If a paramterless method defined as `def $f$: $T$ = ...` *or* `def $f$ = ...` *overrides a method of type $()T'$ which has an empty parameter list, then f is also assumed to have an empty parameter list.*

Another restriction applies to abstract type members: An abstract type member with a volatile type as its upper bound may not override an abstract type member which does not have a volatile upper bound.

An overriding method inherits all default arguments from the definition in the superclass. By specifying default arguments in the overriding method it is possible to add new defaults (if the corresponding parameter in the superclass does not have a default) or to override the defaults of the superclass (otherwise).

(44) *Consider the definitions:*

```
trait Root { type T <: Root }
trait A extends Root { type T <: A }
trait B extends Root { type T <: B }
trait C extends A with B
```

Then the class definition `C` *is not well-formed because the binding of* `T` *in* `C` *is* `type T <: B`*, which fails to subsume the binding* `type T <: A` *of* `T` *in type* `A`*. The problem can be solved by adding an overriding definition of type* `T` *in class* `C`*:*

```
class C extends A with B { type T <: C }
```

5.1.5 Inheritance Closure

Let C be a class type. The inheritance closure *of C is the smallest set $\mathscr{S}$ of types such that*

- *If T is in $\mathscr{S}$, then every type T' which forms syntactically a part of T is also in $\mathscr{S}$.*
- *If T is a class type in $\mathscr{S}$, then all parents of T are also in $\mathscr{S}$.*

It is a static error if the inheritance closure of a class type consists of an infinite number of types. (This restriction is necessary to make subtyping decidable (Kennedy and Pierce 2007)).

5.1.6 Early Definitions

```
EarlyDefs         ::= ‘{’ [EarlyDef {semi EarlyDef}] ‘}’ ‘with’
EarlyDef          ::=  {Annotation} {Modifier} PatVarDef
```

A template may start with an early field definition *clause, which serves to define certain field values before the supertype constructor is called. In a template*

```
{ val $p_1$: $T_1$ = $e_1$
  ...
  val $p_n$: $T_n$ = $e_n$
} with $sc$ with $mt_1$ with $mt_n$ { $\mathit{stats}$ }
```

The initial pattern definitions of $p_1, \ldots, p_n$ *are called* early definitions. *They define fields which form part of the template. Every early definition must define at least one variable.*

An early definition is type-checked and evaluated in the scope which is in effect just before the template being defined, augmented by any type parameters of the enclosing class and by any early definitions preceding the one being defined. In particular, any reference to `this` *in the right-hand side of an early definition refers to the identity of* `this` *just outside the template. Consequently, it is impossible that an early definition refers to the object being constructed by the template, or refers to one of its fields and methods, except for any other preceding early definition in the same section. Furthermore, references to preceding early definitions always refer to the value that's defined there, and do not take into account overriding definitions. In other words, a block of early definitions is evaluated exactly as if it was a local bock containing a number of value definitions.*

Early definitions are evaluated in the order they are being defined before the superclass constructor of the template is called.

(45) Early definitions are particularly useful for traits, which do not have normal constructor parameters. Example:

```
trait Greeting {
  val name: String
  val msg = "How are you, "+name
}
class C extends {
  val name = "Bob"
} with Greeting {
  println(msg)
}
```

In the code above, the field `name` *is initialized before the constructor of* `Greeting` *is called. Therefore, field* `msg` *in class* `Greeting` *is properly initialized to* `"How are you, Bob"`.

If `name` had been initialized instead in `C`'s normal class body, it would be initialized after the constructor of `Greeting`. In that case, `msg` would be initialized to "`How are you, <null>`".

5.2 Modifiers

```
Modifier          ::=  LocalModifier
                    |  AccessModifier
                    |  ‘override’
LocalModifier     ::=  ‘abstract’
                    |  ‘final’
                    |  ‘sealed’
                    |  ‘implicit’
                    |  ‘lazy’
AccessModifier    ::=  (‘private’ | ‘protected’) [AccessQualifier]
AccessQualifier   ::=  ‘[’ (id | ‘this’) ‘]’
```

Member definitions may be preceded by modifiers which affect the accessibility and usage of the identifiers bound by them. If several modifiers are given, their order does not matter, but the same modifier may not occur more than once. Modifiers preceding a repeated definition apply to all constituent definitions. The rules governing the validity and meaning of a modifier are as follows.

- *The `private` modifier can be used with any definition or declaration in a template. Such members can be accessed only from within the directly enclosing template and its companion module or companion class. They are not inherited by subclasses and they may not override definitions in parent classes.*

The modifier can be qualified *with an identifier C (e.g. `private[$C$]`) that must denote a class or package enclosing the definition. Members labeled with such a modifier are accessible respectively only from code inside the package C or only from code inside the class C and its companion module. Such members are also inherited only from templates inside C.*

An different form of qualification is `private[this]`. A member M marked with this modifier can be accessed only from within the object in which it is defined. That is, a selection $p.M$ is only legal if the prefix is `this` or `$O$.this`, for some class O enclosing the reference. In addition, the restrictions for unqualified `private` apply.

Members marked private without a qualifier are called class-private, *whereas members labeled with `private[this]` are called* object-private. *A member* is private *if it is either class-private or object-private, but not if it is marked `private[$C$]` where C is an identifier; in the latter case the member is called* qualified private.

Class-private or object-private members may not be abstract, and may not have `protected` or `override` modifiers.

- *The* `protected` *modifier applies to class member definitions. Protected members of a class can be accessed from within*
  - *the template of the defining class,*
  - *all templates that have the defining class as a base class,*
  - *the companion module of any of those classes. A* `protected` *modifier can be qualified with an identifier C (e.g.* `protected[$C$]`*) that must denote a class or package enclosing the definition. Members labeled with such a modifier are also accessible respectively from all code inside the package C or from all code inside the class C and its companion module.*

A protected identifier x may be used as a member name in a selection `$r$.$x$` *only if one of the following applies: - The access is within the template defining the member, or, if a qualification C is given, inside the package C, or the class C, or its companion module, or - r is one of the reserved words* `this` *and* `super`*, or - r's type conforms to a type-instance of the class which contains the access.*

A different form of qualification is `protected[this]`*. A member M marked with this modifier can be accessed only from within the object in which it is defined. That is, a selection $p.M$ is only legal if the prefix is* `this` *or* `$O$.this`*, for some class O enclosing the reference. In addition, the restrictions for unqualified* `protected` *apply.*

- *The* `override` *modifier applies to class member definitions or declarations. It is mandatory for member definitions or declarations that override some other concrete member definition in a parent class. If an* `override` *modifier is given, there must be at least one overridden member definition or declaration (either concrete or abstract).*
- *The* `override` *modifier has an additional significance when combined with the* `abstract` *modifier. That modifier combination is only allowed for value members of traits.*

We call a member M of a template incomplete *if it is either abstract (i.e. defined by a declaration), or it is labeled* `abstract` *and* `override` *and every member overridden by M is again incomplete.*

Note that the `abstract override` *modifier combination does not influence the concept whether a member is concrete or abstract. A member is* abstract *if only a declaration is given for it; it is* concrete *if a full definition is given.*

- *The* `abstract` *modifier is used in class definitions. It is redundant for traits, and mandatory for all other classes which have incomplete members. Abstract classes cannot be instantiated with a constructor invocation unless followed by mixins and/or a refinement which override all incomplete members of the class. Only abstract classes and traits can have abstract term members.*

The `abstract` *modifier can also be used in conjunction with* `override` *for class member definitions. In that case the previous discussion applies.*

- *The* `final` *modifier applies to class member definitions and to class definitions. A* `final` *class member definition may not be overridden in subclasses. A* `final` *class may not be inherited by a template.* `final` *is redundant for object definitions. Members of final classes or objects are implicitly also final, so the* `final` *modifier is generally redundant for them, too. Note, however, that constant value definitions do require an explicit* `final` *modifier, even if they are defined in a final class or object.* `final` *may not be applied to incomplete members, and it may not be combined in one modifier list with* `sealed`*.*

- *The* `sealed` *modifier applies to class definitions. A* `sealed` *class may not be directly inherited, except if the inheriting template is defined in the same source file as the inherited class. However, subclasses of a sealed class can be inherited anywhere.*

- *The* `lazy` *modifier applies to value definitions. A* `lazy` *value is initialized the first time it is accessed (which might never happen at all). Attempting to access a lazy value during its initialization might lead to looping behavior. If an exception is thrown during initialization, the value is considered uninitialized, and a later access will retry to evaluate its right hand side.*

(46) *The following code illustrates the use of qualified private:*

```
package outerpkg.innerpkg
class Outer {
  class Inner {
    private[Outer] def f()
    private[innerpkg] def g()
    private[outerpkg] def h()
  }
}
```

Here, accesses to the method `f` *can appear anywhere within* `OuterClass`*, but not outside it. Accesses to method* `g` *can appear anywhere within the package* `outerpkg.innerpkg`*, as would be the case for package-private methods in Java. Finally, accesses to method* `h` *can appear anywhere within package* `outerpkg`*, including packages contained in it.*

(47) *A useful idiom to prevent clients of a class from constructing new instances of that class is to declare the class* `abstract` *and* `sealed`*:*

```
object m {
  abstract sealed class C (x: Int) {
    def nextC = new C(x + 1) {}
  }
  val empty = new C(0) {}
}
```

For instance, in the code above clients can create instances of class `m.C` *only by calling the* `nextC` *method of an existing* `m.C` *object; it is not possible for clients to create objects of class* `m.C` *directly. Indeed the following two lines are both in error:*

```
new m.C(0)    // **** error: C is abstract, so it cannot be instantiated.
new m.C(0) {} // **** error: illegal inheritance from sealed class.
```

A similar access restriction can be achieved by marking the primary constructor `private` *(see* **??***).*

5.3 Class Definitions

```
TmplDef           ::=  ‘class’ ClassDef
ClassDef          ::=  id [TypeParamClause] {Annotation}
                     [AccessModifier] ClassParamClauses ClassTemplateOpt
ClassParamClauses ::=  {ClassParamClause}
                       [[nl] ‘(’ implicit ClassParams ‘)’]
ClassParamClause  ::=  [nl] ‘(’ [ClassParams] ’)’
ClassParams       ::=  ClassParam {‘,’ ClassParam}
ClassParam        ::=  {Annotation} [{Modifier} (‘val’ | ‘var’)]
                       id [‘:’ ParamType] [‘=’ Expr]
ClassTemplateOpt ::= ‘extends’ ClassTemplate | [[‘extends’] TemplateBody]
```

The most general form of class definition is

```
class $c$[$\mathit{tps}\,$] $as$ $m$($\mathit{ps}_1$)$\ldots$($\mathit{ps}_n$) extends $t$
```

Here,

- c *is the name of the class to be defined.*
- *tps is a non-empty list of type parameters of the class being defined. The scope of a type parameter is the whole class definition including the type parameter section itself. It is illegal to define two type parameters with the same name. The type parameter section* `[$\mathit{tps}\,$]` *may be omitted. A class with a type parameter section is called* polymorphic*, otherwise it is called* monomorphic.
- as *is a possibly empty sequence of annotations. If any annotations are given, they apply to the primary constructor of the class.*
- m *is an access modifier such as* `private` *or* `protected`*, possibly with a qualification. If such an access modifier is given it applies to the primary constructor to the class.*

- $(ps_1)\ldots(ps_n)$ *are formal value parameter clauses for the* {primary constructor} of the class. The scope of a formal value parameter includes all subsequent parameter sections and the template t. However, a formal value parameter may not form part of the types of any of the parent classes or members of the class template t. It is illegal to define two formal value parameters with the same name. If no formal parameter sections are given, an empty parameter section `()` is assumed.

  If a formal parameter declaration $x : T$ is preceded by a `val` or `var` keyword, an accessor (getter) definition for this parameter is implicitly added to the class. The getter introduces a value member x of class c that is defined as an alias of the parameter. If the introducing keyword is `var`, a setter accessor x_= is also implicitly added to the class. In invocation of that setter `$x$_=($e$)` changes the value of the parameter to the result of evaluating e. The formal parameter declaration may contain modifiers, which then carry over to the accessor definition(s). A formal parameter prefixed by `val` or `var` may not at the same time be a call-by-name parameter.

- t is a template of the form

  ```
  $sc$ with $mt_1$ with $\ldots$ with $mt_m$ { $\mathit{stats}$ } // $m \geq 0$
  ```

  which defines the base classes, behavior and initial state of objects of the class. The extends clause `extends $sc$ with $mt_1$ with $\ldots$ with $mt_m$` can be omitted, in which case `extends scala.AnyRef` is assumed. The class body `{ $\mathit{stats}$ }` may also be omitted, in which case the empty body `{}` is assumed.

This class definition defines a type `$c$[$\mathit{tps}\,$]` and a constructor which when applied to parameters conforming to types ps initializes instances of type `$c$[$\mathit{tps}\,$]` by evaluating the template t.

(48) *The following example illustrates `val` and `var` parameters of a class `C`:*

```
class C(x: Int, val y: String, var z: List[String])
val c = new C(1, "abc", List())
c.z = c.y :: c.z
```

(49) *The following class can be created only from its companion module.*

```
object Sensitive {
  def makeSensitive(credentials: Certificate): Sensitive =
    if (credentials == Admin) new Sensitive()
    else throw new SecurityViolationException
}
class Sensitive private () {
  ...
}
```

5.3.1 Constructor Definitions

```
FunDef         ::= ‘this’ ParamClause ParamClauses
                   (‘=’ ConstrExpr | [nl] ConstrBlock)
ConstrExpr     ::= SelfInvocation
                |  ConstrBlock
ConstrBlock    ::= ‘{’ SelfInvocation {semi BlockStat} ‘}’
SelfInvocation ::= ‘this’ ArgumentExprs {ArgumentExprs}
```

A class may have additional constructors besides the primary constructor. These are defined by constructor definitions of the form `def this($\mathit{ps}_1$)$\ldots$($\mathit{ps}_n$) = $e$`*. Such a definition introduces an additional constructor for the enclosing class, with parameters as given in the formal parameter lists* $ps_1, \ldots, ps_n$*, and whose evaluation is defined by the constructor expression* e*. The scope of each formal parameter is the subsequent parameter sections and the constructor expression* e*. A constructor expression is either a self constructor invocation* `this($\mathit{args}_1$)$\ldots$($\mathit{args}_n$)` *or a block which begins with a self constructor invocation. The self constructor invocation must construct a generic instance of the class. I.e. if the class in question has name* C *and type parameters* `[$\mathit{tps}\,$]`*, then a self constructor invocation must generate an instance of* `$C$[$\mathit{tps}\,$]`*; it is not permitted to instantiate formal type parameters.*

The signature and the self constructor invocation of a constructor definition are type-checked and evaluated in the scope which is in effect at the point of the enclosing class definition, augmented by any type parameters of the enclosing class and by any early definitions of the enclosing template. The rest of the constructor expression is type-checked and evaluated as a function body in the current class.

If there are auxiliary constructors of a class C*, they form together with* C*'s primary constructor an overloaded constructor definition. The usual rules for overloading resolution apply for constructor invocations of* C*, including for the self constructor invocations in the constructor expressions themselves. However, unlike other methods, constructors are never inherited. To prevent infinite cycles of constructor invocations, there is the restriction that every self constructor invocation must refer to a constructor definition which precedes it (i.e. it must refer to either a preceding auxiliary constructor or the primary constructor of the class).*

(50) *Consider the class definition*

```scala
class LinkedList[A]() {
  var head = _
  var tail = null
  def isEmpty = tail != null
  def this(head: A) = { this(); this.head = head }
  def this(head: A, tail: List[A]) = { this(head); this.tail = tail }
}
```

> *This defines a class `LinkedList` with three constructors. The second constructor constructs an singleton list, while the third one constructs a list with a given head and tail.*

5.4 Case Classes

```
TmplDef  ::=  'case' 'class' ClassDef
```

If a class definition is prefixed with `case`, the class is said to be a case class.

The formal parameters in the first parameter section of a case class are called elements*; they are treated specially. First, the value of such a parameter can be extracted as a field of a constructor pattern. Second, a `val` prefix is implicitly added to such a parameter, unless the parameter carries already a `val` or `var` modifier. Hence, an accessor definition for the parameter is generated.*

A case class definition of `$c$[$\mathit{tps}\,$]($\mathit{ps}_1\,$)$\ldots$($\mathit{ps}_n$)` with type parameters tps and value parameters ps implicitly generates an extractor object which is defined as follows:

```
object $c$ {
  def apply[$\mathit{tps}\,$]($\mathit{ps}_1\,$)$\ldots$($\mathit{ps}_n$): $c$[$\mathit{tps}\,$] = new
  def unapply[$\mathit{tps}\,$]($x$: $c$[$\mathit{tps}\,$]) =
    if (x eq null) scala.None
    else scala.Some($x.\mathit{xs}_{11}, \ldots , x.\mathit{xs}_{1k}$)
}
```

Here, Ts stands for the vector of types defined in the type parameter section tps, each xs_i denotes the parameter names of the parameter section ps_i, and $xs_{11}, \ldots, xs_{1k}$ denote the names of all parameters in the first parameter section xs_1. If a type parameter section is missing in the class, it is also missing in the `apply` and `unapply` methods. The definition of `apply` is omitted if class c is `abstract`.

If the case class definition contains an empty value parameter list, the `unapply` method returns a `Boolean` instead of an `Option` type and is defined as follows:

```
def unapply[$\mathit{tps}\,$]($x$: $c$[$\mathit{tps}\,$]) = x ne null
```

The name of the `unapply` method is changed to `unapplySeq` if the first parameter section ps_1 of c ends in a repeated parameter. If a companion object c exists already, no new object is created, but the `apply` and `unapply` methods are added to the existing object instead.

A method named `copy` is implicitly added to every case class unless the class already has a member (directly defined or inherited) with that name. The method is defined as follows:

```
def copy[$\mathit{tps}\,$]($\mathit{ps}'_1\,$)$\ldots$($\mathit{ps}'_n$): $c$[$\mathit{tps}
```

Again, Ts stands for the vector of types defined in the type parameter section tps and each xs_i denotes the parameter names of the parameter section ps'_i. Every value parameter $ps'_{i,j}$ of the `copy` *method has the form* `$x_{i,j}$:$T_{i,j}$=this.$x_{i,j}$`*, where $x_{i,j}$ and $T_{i,j}$ refer to the name and type of the corresponding class parameter $ps_{i,j}$.*

Every case class implicitly overrides some method definitions of class `scala.AnyRef` *unless a definition of the same method is already given in the case class itself or a concrete definition of the same method is given in some base class of the case class different from* `AnyRef`*. In particular:*

- *Method* `equals: (Any)Boolean` *is structural equality, where two instances are equal if they both belong to the case class in question and they have equal (with respect to* `equals`*) constructor arguments.*
- *Method* `hashCode: Int` *computes a hash-code. If the hashCode methods of the data structure members map equal (with respect to equals) values to equal hash-codes, then the case class hashCode method does too.*
- *Method* `toString: String` *returns a string representation which contains the name of the class and its elements.*

(51) *Here is the definition of abstract syntax for lambda calculus:*

```
class Expr
case class Var    (x: String)          extends Expr
case class Apply  (f: Expr, e: Expr)   extends Expr
case class Lambda(x: String, e: Expr) extends Expr
```

This defines a class `Expr` *with case classes* `Var`*,* `Apply` *and* `Lambda`*. A call-by-value evaluator for lambda expressions could then be written as follows.*

```
type Env = String => Value
case class Value(e: Expr, env: Env)

def eval(e: Expr, env: Env): Value = e match {
  case Var (x) =>
    env(x)
  case Apply(f, g) =>
    val Value(Lambda (x, e1), env1) = eval(f, env)
    val v = eval(g, env)
    eval (e1, (y => if (y == x) v else env1(y)))
  case Lambda(_, _) =>
    Value(e, env)
}
```

It is possible to define further case classes that extend type `Expr` *in other parts of the program, for instance*

```
case class Number(x: Int) extends Expr
```

This form of extensibility can be excluded by declaring the base class `Expr` `sealed`; in this case, all classes that directly extend `Expr` must be in the same source file as `Expr`.

5.4.1 Traits

```
TmplDef          ::=  ‘trait’ TraitDef
TraitDef         ::=  id [TypeParamClause] TraitTemplateOpt
TraitTemplateOpt ::=  ‘extends’ TraitTemplate | [[‘extends’] TemplateBody]
```

A trait is a class that is meant to be added to some other class as a mixin. Unlike normal classes, traits cannot have constructor parameters. Furthermore, no constructor arguments are passed to the superclass of the trait. This is not necessary as traits are initialized after the superclass is initialized.

Assume a trait D defines some aspect of an instance x of type C (i.e. D is a base class of C). Then the {actual supertype} of D in x is the compound type consisting of all the base classes in $\mathcal{L}(C)$ that succeed D. The actual supertype gives the context for resolving a `super` reference in a trait. Note that the actual supertype depends on the type to which the trait is added in a mixin composition; it is not statically known at the time the trait is defined.

If D is not a trait, then its actual supertype is simply its least proper supertype (which is statically known).

(52) The following trait defines the property of being comparable to objects of some type. It contains an abstract method `<` and default implementations of the other comparison operators `<=`, `>`, and `>=`.

```
trait Comparable[T <: Comparable[T]] { self: T =>
  def < (that: T): Boolean
  def <=(that: T): Boolean = this < that || this == that
  def > (that: T): Boolean = that < this
  def >=(that: T): Boolean = that <= this
}
```

(53) Consider an abstract class `Table` that implements maps from a type of keys `A` to a type of values `B`. The class has a method `set` to enter a new key / value pair into the table, and a method `get` that returns an optional value matching a given key. Finally, there is a method `apply` which is like `get`, except that it returns a given default value if the table is undefined for the given key. This class is implemented as follows.

```
abstract class Table[A, B](defaultValue: B) {
  def get(key: A): Option[B]
```

```
  def set(key: A, value: B)
  def apply(key: A) = get(key) match {
    case Some(value) => value
    case None => defaultValue
  }
}
```

Here is a concrete implementation of the Table class.

```
class ListTable[A, B](defaultValue: B) extends Table[A, B](defaultValue) {
  private var elems: List[(A, B)]
  def get(key: A) = elems.find(._1.==(key)).map(._2)
  def set(key: A, value: B) = { elems = (key, value) :: elems }
}
```

Here is a trait that prevents concurrent access to the get and set operations of its parent class:

```
trait SynchronizedTable[A, B] extends Table[A, B] {
  abstract override def get(key: A): B =
    synchronized { super.get(key) }
  abstract override def set((key: A, value: B) =
    synchronized { super.set(key, value) }
}
```

Note that SynchronizedTable does not pass an argument to its superclass, Table, even though Table is defined with a formal parameter. Note also that the super calls in SynchronizedTable's get and set methods statically refer to abstract methods in class Table. This is legal, as long as the calling method is labeled abstract override.

Finally, the following mixin composition creates a synchronized list table with strings as keys and integers as values and with a default value 0:

```
object MyTable extends ListTable[String, Int](0) with SynchronizedTable
```

The object MyTable inherits its get and set method from SynchronizedTable. The super calls in these methods are re-bound to refer to the corresponding implementations in ListTable, which is the actual supertype of SynchronizedTable in MyTable.

5.5 Object Definitions

```
ObjectDef       ::=  id ClassTemplate
```

An object definition defines a single object of a new class. Its most general form is `object $m$ extends $t$`. Here, m is the name of the object to be defined, and t is a template of the form

```
$sc$ with $mt_1$ with $\ldots$ with $mt_n$ { $\mathit{stats}$ }
```

which defines the base classes, behavior and initial state of m. The extends clause `extends $sc$ with $mt_1$ with $\ldots$ with $mt_n$` can be omitted, in which case `extends scala.AnyRef` is assumed. The class body `{ $\mathit{stats}$ }` may also be omitted, in which case the empty body `{}` is assumed.

The object definition defines a single object (or: *module*) conforming to the template t. It is roughly equivalent to the following definition of a lazy value:

```
lazy val $m$ = new $sc$ with $mt_1$ with $\ldots$ with $mt_n$ { this: $m.type$ => $\mathit{stats}$ }
```

Note that the value defined by an object definition is instantiated lazily. The `new $m\Dollar$cls` constructor is evaluated not at the point of the object definition, but is instead evaluated the first time m is dereferenced during execution of the program (which might be never at all). An attempt to dereference m again in the course of evaluation of the constructor leads to a infinite loop or run-time error.
Other threads trying to dereference m while the constructor is being evaluated block until evaluation is complete.

The expansion given above is not accurate for top-level objects. It cannot be because variable and method definition cannot appear on the top-level outside of a package object. Instead, top-level objects are translated to static fields.

(54) Classes in Scala do not have static members; however, an equivalent effect can be achieved by an accompanying object definition E.g.

```
abstract class Point {
  val x: Double
  val y: Double
  def isOrigin = (x == 0.0 && y == 0.0)
}
object Point {
  val origin = new Point() { val x = 0.0; val y = 0.0 }
}
```

This defines a class `Point` and an object `Point` which contains `origin` as a member. Note that the double use of the name `Point` is legal, since the class definition defines the name `Point` in the type name space, whereas the object definition defines a name in the term namespace.

This technique is applied by the Scala compiler when interpreting a Java class with static members. Such a class C is conceptually seen as a pair of a Scala class that contains all instance members of C and a Scala object that contains all static members of C.

Generally, a *companion module* of a class is an object which has the same name as the class and is defined in the same scope and compilation unit. Conversely, the class is called the *companion class* of the module.

Chapter 6

Expressions

```
Expr            ::=  (Bindings | id | ‘_’) ‘=>’ Expr
                  |  Expr1
Expr1           ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] else Expr]
                  |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
                |  ‘try’ ‘{’ Block ‘}’ [‘catch’  ‘{’ CaseClauses ‘}’]
                     [‘finally’ Expr]
                  |  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ’)’
                 |  ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’)
                     {nl} [‘yield’] Expr
                  |  ‘throw’ Expr
                  |  ‘return’ [Expr]
                  |  [SimpleExpr ‘.’] id ‘=’ Expr
                  |  SimpleExpr1 ArgumentExprs ‘=’ Expr
                  |  PostfixExpr
                  |  PostfixExpr Ascription
                  |  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
PostfixExpr     ::=  InfixExpr [id [nl]]
InfixExpr       ::=  PrefixExpr
                  |  InfixExpr id [nl] InfixExpr
PrefixExpr      ::=  [‘-’ | ‘+’ | ‘~’ | ‘!’] SimpleExpr
SimpleExpr      ::=  ‘new’ (ClassTemplate | TemplateBody)
                  |  BlockExpr
                  |  SimpleExpr1 [‘_’]
SimpleExpr1     ::=  Literal
                  |  Path
                  |  ‘_’
                  |  ‘(’ [Exprs] ‘)’
                  |  SimpleExpr ‘.’ id s
                  |  SimpleExpr TypeArgs
                  |  SimpleExpr1 ArgumentExprs
```

```
                      |  XmlExpr
Exprs                ::=  Expr {‘,’ Expr}
BlockExpr            ::=  ‘{’ CaseClauses ‘}’
                      |  ‘{’ Block ‘}’
Block                ::=  {BlockStat semi} [ResultExpr]
ResultExpr           ::=  Expr1
               |  (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block
Ascription           ::=  ‘:’ InfixType
                      |  ‘:’ Annotation {Annotation}
                      |  ‘:’ ‘_’ ‘*’
```

Expressions are composed of operators and operands. Expression forms are discussed subsequently in decreasing order of precedence.

6.1 Expression Typing

The typing of expressions is often relative to some *expected type* (which might be undefined). When we write “expression e is expected to conform to type T”, we mean: (1) the expected type of e is T, and (2) the type of expression e must conform to T.

The following skolemization rule is applied universally for every expression: If the type of an expression would be an existential type T, then the type of the expression is assumed instead to be a skolemization of T.

Skolemization is reversed by type packing. Assume an expression e of type T and let $t_1[\mathit{tps}_1] >: L_1 <: U_1, \ldots, t_n[\mathit{tps}_n] >: L_n <: U_n$ be all the type variables created by skolemization of some part of e which are free in T. Then the *packed type* of e is

```
$T$ forSome { type $t_1[\mathit{tps}_1] >: L_1 <: U_1$; $\ldots$; type $t_n[\mathit{tps}_n]
```

6.2 Literals

```
SimpleExpr    ::=  Literal
```

Typing of literals is as described here; their evaluation is immediate.

6.3 The *Null* Value

The `null` value is of type `scala.Null`, and is thus compatible with every reference type. It denotes a reference value which refers to a special “`null`” object. This object implements methods in class `scala.AnyRef` as follows:

- `eq($x\,$)` and `==($x\,$)` return `true` iff the argument x is also the "null" object.
- `ne($x\,$)` and `!=($x\,$)` return true iff the argument x is not also the "null" object.
- `isInstanceOf[$T\,$]` always returns `false`.
- `asInstanceOf[$T\,$]` returns the 'null`" object itself if $T$ conforms` toscala.AnyRef`, and throws` aNullPointerException' otherwise.

A reference to any other member of the 'null`" object causes` aNullPointerException' to be thrown.

6.4 Designators

```
SimpleExpr  ::=  Path
              |  SimpleExpr '.' id
```

A designator refers to a named term. It can be a *simple name* or a *selection*.

A simple name x refers to a value as specified here. If x is bound by a definition or declaration in an enclosing class or object C, it is taken to be equivalent to the selection `$C$.this.$x$` where C is taken to refer to the class containing x even if the type name C is shadowed at the occurrence of x.

If r is a stable identifier of type T, the selection $r.x$ refers statically to a term member m of r that is identified in T by the name x.

For other expressions e, $e.x$ is typed as if it was `{ val $y$ = $e$; $y$.$x$ }`, for some fresh name y.

The expected type of a designator's prefix is always undefined. The type of a designator is the type T of the entity it refers to, with the following exception: The type of a path p which occurs in a context where a stable type is required is the singleton type `$p$.type`.

The contexts where a stable type is required are those that satisfy one of the following conditions:

1. The path p occurs as the prefix of a selection and it does not designate a constant, or
2. The expected type *pt* is a stable type, or
3. The expected type *pt* is an abstract type with a stable type as lower bound, and the type T of the entity referred to by p does not conform to *pt*, or
4. The path p designates a module.

The selection $e.x$ is evaluated by first evaluating the qualifier expression e, which yields an object r, say. The selection's result is then the member of r that is either defined by m or defined by a definition overriding m. If that member has a type which conforms to `scala.NotNull`, the member's value must be initialized to a value different from `null`, otherwise a `scala.UnitializedError` is thrown.

6.5 This and Super

```
SimpleExpr  ::=  [id ‘.’] ‘this’
              |  [id ’.’] ‘super’ [ClassQualifier] ‘.’ id
```

The expression `this` can appear in the statement part of a template or compound type. It stands for the object being defined by the innermost template or compound type enclosing the reference. If this is a compound type, the type of `this` is that compound type. If it is a template of a class or object definition with simple name C, the type of this is the same as the type of `$C$.this`.

The expression `$C$.this` is legal in the statement part of an enclosing class or object definition with simple name C. It stands for the object being defined by the innermost such definition. If the expression's expected type is a stable type, or `$C$.this` occurs as the prefix of a selection, its type is `$C$.this.type`, otherwise it is the self type of class C.

A reference `super.$m$` refers statically to a method or type m in the least proper supertype of the innermost template containing the reference. It evaluates to the member m' in the actual supertype of that template which is equal to m or which overrides m. The statically referenced member m must be a type or a method.

If it is a method, it must be concrete, or the template containing the reference must have a member m' which overrides m and which is labeled `abstract override`.

A reference `$C$.super.$m$` refers statically to a method or type m in the least proper supertype of the innermost enclosing class or object definition named C which encloses the reference. It evaluates to the member m' in the actual supertype of that class or object which is equal to m or which overrides m. The statically referenced member m must be a type or a method. If the statically referenced member m is a method, it must be concrete, or the innermost enclosing class or object definition named C must have a member m' which overrides m and which is labeled `abstract override`.

The super prefix may be followed by a trait qualifier `[$T\,$]`, as in `$C$.super[$T\,$].$x$`. This is called a *static super reference*. In this case, the reference is to the type or method of x in the parent trait of C whose simple name is T. That member must be uniquely defined. If it is a method, it must be concrete.

(55) Consider the following class definitions

```
class Root { def x = "Root" }
class A extends Root { override def x = "A" ; def superA = super.x }
trait B extends Root { override def x = "B" ; def superB = super.x }
class C extends Root with B {
  override def x = "C" ; def superC = super.x
}
class D extends A with B {
  override def x = "D" ; def superD = super.x
}
```

The linearization of class `C` is `{C, B, Root}` and the linearization of class `D` is `{D, B, A, Root}`. Then we have:

```
(new A).superA == "Root",
                              (new C).superB = "Root", (new C).superC = "B",
(new D).superA == "Root", (new D).superB = "A",    (new D).superD = "B",
```

Note that the `superB` function returns different results depending on whether `B` is mixed in with class `Root` or `A`.

6.6 Function Applications

```
SimpleExpr    ::=  SimpleExpr1 ArgumentExprs
ArgumentExprs ::=  '(' [Exprs] ')'
                |  '(' [Exprs ','] PostfixExpr ':' '_' '*' ')'
                |  [nl] BlockExpr
Exprs         ::=  Expr {',' Expr}
```

An application `$f$($e_1 , \ldots , e_m$)` applies the function f to the argument expressions $e_1, \ldots, e_m$. If f has a method type `($p_1$:$T_1 , \ldots , p_n$:$T_n$)$U$`, the type of each argument expression e_i is typed with the corresponding parameter type T_i as expected type. Let S_i be type type of argument e_i $(i = 1, \ldots, m)$. If f is a polymorphic method, local type inference is used to determine type arguments for f. If f has some value type, the application is taken to be equivalent to `$f$.apply($e_1 , \ldots , e_m$)`, i.e. the application of an `apply` method defined by f.

The function f must be *applicable* to its arguments $e_1, \ldots, e_n$ of types $S_1, \ldots, S_n$.

If f has a method type $(p_1 : T_1, \ldots, p_n : T_n)U$ we say that an argument expression e_i is a *named* argument if it has the form $x_i = e'_i$ and x_i is one of the parameter names $p_1, \ldots, p_n$. The function f is applicable if all of the follwing conditions hold:

- For every named argument $x_i = e'_i$ the type S_i is compatible with the parameter type T_j whose name p_j matches x_i.
- For every positional argument e_i the type S_i is compatible with T_i.
- If the expected type is defined, the result type U is compatible to it.

If f is a polymorphic method it is applicable if local type inference can determine type arguments so that the instantiated method is applicable. If f has some value type it is applicable if it has a method member named `apply` which is applicable.

Evaluation of `$f$($e_1 , \ldots , e_n$)` usually entails evaluation of f and $e_1, \ldots, e_n$ in that order. Each argument expression is converted to the type of its corresponding formal parameter. After that, the application is rewritten to the function's right hand side, with actual arguments substituted for formal parameters. The result of evaluating the rewritten right-hand side is finally converted to the function's declared result type, if one is given.

The case of a formal parameter with a parameterless method type `=>$T$` is treated specially. In this case, the corresponding actual argument expression e is not evaluated before the application. Instead, every use of the formal parameter on the right-hand side of the rewrite rule entails a re-evaluation of e. In other words, the evaluation order for `=>`-parameters is *call-by-name* whereas the evaluation order for normal parameters is *call-by-value*. Furthermore, it is required that e's packed type conforms to the parameter type T. The behavior of by-name parameters is preserved if the application is transformed into a block due to named or default arguments. In this case, the local value for that parameter has the form `val $y_i$ = () => $e$` and the argument passed to the function is `$y_i$()`.

The last argument in an application may be marked as a sequence argument, e.g. `$e$: _*`. Such an argument must correspond to a repeated parameter of type `$S$*` and it must be the only argument matching this parameter (i.e. the number of formal parameters and actual arguments must be the same). Furthermore, the type of e must conform to `scala.Seq[$T$]`, for some type T which conforms to S. In this case, the argument list is transformed by replacing the sequence e with its elements. When the application uses named arguments, the vararg parameter has to be specified exactly once.

A function application usually allocates a new frame on the program's run-time stack. However, if a local function or a final method calls itself as its last action, the call is executed using the stack-frame of the caller.

(56) Assume the following function which computes the sum of a variable number of arguments:

```
def sum(xs: Int*) = (0 /: xs) ((x, y) => x + y)
```

Then

```
sum(1, 2, 3, 4)
sum(List(1, 2, 3, 4): _*)
```

both yield 10 as result. On the other hand,

```
sum(List(1, 2, 3, 4))
```

would not typecheck.

6.6.1 Named and Default Arguments

If an application might uses named arguments $p = e$ or default arguments, the following conditions must hold.

- The named arguments form a suffix of the argument list $e_1, \ldots, e_m$, i.e. no positional argument follows a named one.
- The names x_i of all named arguments are pairwise distinct and no named argument defines a parameter which is already specified by a positional argument.
- Every formal parameter $p_j : T_j$ which is not specified by either a positional or a named argument has a default argument.

If the application uses named or default arguments the following transformation is applied to convert it into an application without named or default arguments.

If the function f has the form `$p.m$[$\mathit{targs}$]` it is transformed into the block

```
{ val q = $p$
  q.$m$[$\mathit{targs}$]
}
```

If the function f is itself an application expression the transformation is applied recursively on f. The result of transforming f is a block of the form

```
{ val q = $p$
  val $x_1$ = expr$_1$
  $\ldots$
  val $x_k$ = expr$_k$
  q.$m$[$\mathit{targs}$]($\mathit{args}_1$)$, \ldots ,$($\mathit{args}_l$)
}
```

where every argument in $(\mathit{args}_1), \ldots, (\mathit{args}_l)$ is a reference to one of the values $x_1, \ldots, x_k$. To integrate the current application into the block, first a value definition using a fresh name y_i is created for every argument in $e_1, \ldots, e_m$, which is initialised to e_i for positional arguments and to e'_i for named arguments of the form `$x_i=e'_i$`. Then, for every parameter which is not specified by the argument list, a value definition using a fresh name z_i is created, which is initialized using the method computing the default argument of this parameter.

Let *args* be a permutation of the generated names y_i and z_i such such that the position of each name matches the position of its corresponding parameter in the method type `($p_1:T_1 , \ldots , p_n:T_n$)$U$`. The final result of the transformation is a block of the form

```
{ val q = $p$
  val $x_1$ = expr$_1$
  $\ldots$
  val $x_l$ = expr$_k$
  val $y_1$ = $e_1$
  $\ldots$
```

```
  val $y_m$ = $e_m$
  val $z_1$ = q.$m$\$default\$i[$\mathit{targs}$]($\mathit{args}_1$)$, \ldots ,$($\mathit{a
  $\ldots$
  val $z_d$ = q.$m$\$default\$j[$\mathit{targs}$]($\mathit{args}_1$)$, \ldots ,$($\mathit{a
  q.$m$[$\mathit{targs}$]($\mathit{args}_1$)$, \ldots ,$($\mathit{args}_l$)($\mathit{args}$
}
```

6.7 Method Values

```
SimpleExpr    ::=  SimpleExpr1 '_'
```

The expression `$e$ _` is well-formed if e is of method type or if e is a call-by-name parameter. If e is a method with parameters, `$e$ _~` represents e converted to a function type by eta expansion. If e is a parameterless method or call-by-name parameter of type `=>$T$`, `$e$ _~` represents the function of type `() => $T$`, which evaluates e when it is applied to the empty parameterlist `()`.

(57) The method values in the left column are each equivalent to the anonymous functions on their right.

`Math.sin _`	`x => Math.sin(x)`
`Array.range _`	`(x1, x2) => Array.range(x1, x2)`
`List.map2 _`	`(x1, x2) => (x3) => List.map2(x1, x2)(x3)`
`List.map2(xs, ys)_`	`x => List.map2(xs, ys)(x)`

Note that a space is necessary between a method name and the trailing underscore because otherwise the underscore would be considered part of the name.

6.8 Type Applications

```
SimpleExpr    ::=  SimpleExpr TypeArgs
```

A type application `$e$[$T_1 , \ldots , T_n$]` instantiates a polymorphic value e of type `[$a_1$ >: $L_1$ <: $U_1, \ldots , a_n$ >: $L_n$ <: $U_n$]$S$` with argument types `$T_1 , \ldots , T_n$`. Every argument type T_i must obey the corresponding bounds L_i and U_i. That is, for each $i = 1, \ldots, n$, we must have $\sigma L_i <: T_i <: \sigma U_i$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$. The type of the application is σS.

If the function part e is of some value type, the type application is taken to be equivalent to `$e$.apply[$T_1 , \ldots ,$ T$_n$]`, i.e. the application of an `apply` method defined by e.

Type applications can be omitted if local type inference can infer best type parameters for a polymorphic functions from the types of the actual function arguments and the expected result type.

6.9 Tuples

```
SimpleExpr   ::=  ‘(’ [Exprs] ‘)’
```

A tuple expression `($e_1 , \ldots , e_n$)` is an alias for the class instance creation `scala.Tuple$n$($e_1 , \ldots , e_n$)`, where $n \geq 2$.
The empty tuple `()` is the unique value of type `scala.Unit`.

6.10 Instance Creation Expressions

```
SimpleExpr     ::=  ‘new’ (ClassTemplate | TemplateBody)
```

A simple instance creation expression is of the form `new $c$` where c is a constructor invocation. Let T be the type of c. Then T must denote a (a type instance of) a non-abstract subclass of `scala.AnyRef`. Furthermore, the *concrete self type* of the expression must conform to the self type of the class denoted by T. The concrete self type is normally T, except if the expression `new $c$` appears as the right hand side of a value definition

```
val $x$: $S$ = new $c$
```

(where the type annotation `: $S$` may be missing). In the latter case, the concrete self type of the expression is the compound type `$T$ with $x$.type`.

The expression is evaluated by creating a fresh object of type T which is is initialized by evaluating c. The type of the expression is T.

A general instance creation expression is of the form `new $t$` for some class template t. Such an expression is equivalent to the block

```
{ class $a$ extends $t$; new $a$ }
```

where a is a fresh name of an *anonymous class* which is inaccessible to user programs.

There is also a shorthand form for creating values of structural types: If `{$D$}` is a class body, then `new {$D$}` is equivalent to the general instance creation expression `new AnyRef{$D$}`.

(58) Consider the following structural instance creation expression:

```
new { def getName() = "aaron" }
```

This is a shorthand for the general instance creation expression

```
new AnyRef{ def getName() = "aaron" }
```

The latter is in turn a shorthand for the block

```
{ class anon\$X extends AnyRef{ def getName() = "aaron" }; new anon\$X }
```

where `anon\$X` is some freshly created name.

6.11 Blocks

```
BlockExpr   ::=  '{' Block '}'
Block       ::=  {BlockStat semi} [ResultExpr]
```

A block expression `{$s_1$; $\ldots$; $s_n$; $e\,$}` is constructed from a sequence of block statements $s_1, \ldots, s_n$ and a final expression e. The statement sequence may not contain two definitions or declarations that bind the same name in the same namespace. The final expression can be omitted, in which case the unit value `()` is assumed.

The expected type of the final expression e is the expected type of the block. The expected type of all preceding statements is undefined.

The type of a block `$s_1$; $\ldots$; $s_n$; $e$` is `$T$ forSome {$\,Q\,$}`, where T is the type of e and Q contains existential clauses for every value or type name which is free in T and which is defined locally in one of the statements $s_1, \ldots, s_n$. We say the existential clause *binds* the occurrence of the value or type name. Specifically,

- A locally defined type definition `type$\;t = T$` is bound by the existential clause `type$\;t >: T <: T$`. It is an error if t carries type parameters.
- A locally defined value definition~ `val$\;x: T = e$` is bound by the existential clause `val$\;x: T$`.
- A locally defined class definition `class$\;c$ extends$\;t$` is bound by the existential clause `type$\;c <: T$` where T is the least class type or refinement type which is a proper supertype of the type c. It is an error if c carries type parameters.
- A locally defined object definition `object$\;x\;$extends$\;t$` is bound by the existential clause `val$\;x: T$` where T is the least class type or refinement type which is a proper supertype of the type `$x$.type`.

Evaluation of the block entails evaluation of its statement sequence, followed by an evaluation of the final expression e, which defines the result of the block.

(59) Assuming a class `Ref[T](x: T)`, the block

```
{ class C extends B {$\ldots$} ; new Ref(new C) }
```

has the type `Ref[_1] forSome { type _1 <: B }`. The block

```
{ class C extends B {$\ldots$} ; new C }
```

simply has type `B`, because with the rules here the existentially quantified type `_1 forSome { type _1 <: B }` can be simplified to `B`.

6.12 Prefix, Infix, and Postfix Operations

```
PostfixExpr     ::=  InfixExpr [id [nl]]
InfixExpr       ::=  PrefixExpr
                  |  InfixExpr id [nl] InfixExpr
PrefixExpr      ::=  [‘-’ | ‘+’ | ‘!’ | ‘~’] SimpleExpr
```

Expressions can be constructed from operands and operators.

6.12.1 Prefix Operations

A prefix operation *op*; e consists of a prefix operator *op*, which must be one of the identifiers ‘+’, ‘-’, ‘`!`’ or ‘~’. The expression *op*; e is equivalent to the postfix method application `e.unary_$\mathit{op}$`.

Prefix operators are different from normal function applications in that their operand expression need not be atomic. For instance, the input sequence `-sin(x)` is read as `-(sin(x))`, whereas the function application `negate sin(x)` would be parsed as the application of the infix operator `sin` to the operands `negate` and `(x)`.

6.12.2 Postfix Operations

A postfix operator can be an arbitrary identifier. The postfix operation e; *op* is interpreted as e.*op*.

6.12.3 Infix Operations

An infix operator can be an arbitrary identifier. Infix operators have precedence and associativity defined as follows:

The *precedence* of an infix operator is determined by the operator's first character. Characters are listed below in increasing order of precedence, with characters on the same line having the same precedence.

```
$\mbox{\rm\sl(all letters)}$
|
^
&
< >
= !
:
+ -
* / %
$\mbox{\rm\sl(all other special characters)}$
```

That is, operators starting with a letter have lowest precedence, followed by operators starting with '`|`', etc.

There's one exception to this rule, which concerns *assignment operators*. The precedence of an assigment operator is the same as the one of simple assignment `(=)`. That is, it is lower than the precedence of any other operator.

The *associativity* of an operator is determined by the operator's last character. Operators ending in a colon '`:`' are right-associative. All other operators are left-associative.

Precedence and associativity of operators determine the grouping of parts of an expression as follows.

- If there are several infix operations in an expression, then operators with higher precedence bind more closely than operators with lower precedence.
- If there are consecutive infix operations $e_0; \mathit{op}_1; e_1; \mathit{op}_2 \ldots \mathit{op}_n; e_n$ with operators $\mathit{op}_1, \ldots, \mathit{op}_n$ of the same precedence, then all these operators must have the same associativity. If all operators are left-associative, the sequence is interpreted as $(\ldots(e_0; \mathit{op}_1; e_1); \mathit{op}_2 \ldots); \mathit{op}_n; e_n$. Otherwise, if all operators are right-associative, the sequence is interpreted as $e_0; \mathit{op}_1; (e_1; \mathit{op}_2; (\ldots \mathit{op}_n; e_n) \ldots)$.
- Postfix operators always have lower precedence than infix operators. E.g. $e_1; \mathit{op}_1; e_2; \mathit{op}_2$ is always equivalent to $(e_1; \mathit{op}_1; e_2); \mathit{op}_2$.

The right-hand operand of a left-associative operator may consist of several arguments enclosed in parentheses, e.g. $e; \mathit{op}; (e_1, \ldots, e_n)$. This expression is then interpreted as $e.\mathit{op}(e_1, \ldots, e_n)$.

A left-associative binary operation $e_1; \mathit{op}; e_2$ is interpreted as $e_1.\mathit{op}(e_2)$. If *op* is right-associative, the same operation is interpreted as `{ val $x$=$e_1$; $e_2$.$\mathit{op}$($x\,$) }`, where x is a fresh name.

6.12.4 Assignment Operators

An assignment operator is an operator symbol (syntax category `op` in Identifiers) that ends in an equals character "`=`", with the exception of operators for which one of the following conditions holds:

1. the operator also starts with an equals character, or
2. the operator is one of `(<=)`, `(>=)`, `(!=)`.

Assignment operators are treated specially in that they can be expanded to assignments if no other interpretation is valid.

Let's consider an assignment operator such as += in an infix operation `$l$ += $r$`, where l, r are expressions.
This operation can be re-interpreted as an operation which corresponds to the assignment

```
$l$ = $l$ + $r$
```

except that the operation's left-hand-side l is evaluated only once.

The re-interpretation occurs if the following two conditions are fulfilled.

1. The left-hand-side l does not have a member named +=, and also cannot be converted by an implicit conversion to a value with a member named +=.
2. The assignment `$l$ = $l$ + $r$` is type-correct. In particular this implies that l refers to a variable or object that can be assigned to, and that is convertible to a value with a member named +.

6.13 Typed Expressions

```
Expr1              ::=  PostfixExpr ‘:’ CompoundType
```

The typed expression $e : T$ has type T. The type of expression e is expected to conform to T. The result of the expression is the value of e converted to type T.

(60) Here are examples of well-typed and illegally typed expressions.

```
1: Int               // legal, of type Int
1: Long              // legal, of type Long
// 1: string         // ***** illegal
```

6.14 Annotated Expressions

```
Expr1              ::=  PostfixExpr ‘:’ Annotation {Annotation}
```

An annotated expression `$e$: @$a_1$ $\ldots$ @$a_n$` attaches annotations $a_1, \ldots, a_n$ to the expression e.

6.15 Assignments

```
Expr1         ::=  [SimpleExpr '.'] id '=' Expr
                |  SimpleExpr1 ArgumentExprs '=' Expr
```

The interpretation of an assignment to a simple variable `$x$ = $e$` depends on the definition of x. If x denotes a mutable variable, then the assignment changes the current value of x to be the result of evaluating the expression e. The type of e is expected to conform to the type of x. If x is a parameterless function defined in some template, and the same template contains a setter function `$x$_=` as member, then the assignment `$x$ = $e$` is interpreted as the invocation `$x$_=($e\,$)` of that setter function. Analogously, an assignment `$f.x$ = $e$` to a parameterless function x is interpreted as the invocation `$f.x$_=($e\,$)`.

An assignment `$f$($\mathit{args}\,$) = $e$` with a function application to the left of the '=' operator is interpreted as `$f.$update($\mathit{args}$, $e\,$)`, i.e. the invocation of an `update` function defined by f.

(61) Here are some assignment expressions and their equivalent expansions.

`x.f = e`	x.f_=(e)
`x.f() = e`	x.f.update(e)
`x.f(i) = e`	x.f.update(i, e)
`x.f(i, j) = e`	x.f.update(i, j, e)

(62) Here is the usual imperative code for matrix multiplication.

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length, yss(0).length)
  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss(i)(k) * yss(k)(j)
        k += 1
      }
      zss(i)(j) = acc
      j += 1
    }
    i += 1
```

```
  }
  zss
}
```

Desugaring the array accesses and assignments yields the following expanded version:

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val zss: Array[Array[Double]] = new Array(xss.length, yss.apply(0).length)
  var i = 0
  while (i < xss.length) {
    var j = 0
    while (j < yss.apply(0).length) {
      var acc = 0.0
      var k = 0
      while (k < yss.length) {
        acc = acc + xss.apply(i).apply(k) * yss.apply(k).apply(j)
        k += 1
      }
      zss.apply(i).update(j, acc)
      j += 1
    }
    i += 1
  }
  zss
}
```

6.16 Conditional Expressions

```
Expr1          ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] ‘else’ Expr]
```

The conditional expression `if ($e_1$) $e_2$ else $e_3$` chooses one of the values of e_2 and e_3, depending on the value of e_1. The condition e_1 is expected to conform to type `Boolean`. The then-part e_2 and the else-part e_3 are both expected to conform to the expected type of the conditional expression. The type of the conditional expression is the weak least upper bound of the types of e_2 and e_3. A semicolon preceding the `else` symbol of a conditional expression is ignored.

The conditional expression is evaluated by evaluating first e_1. If this evaluates to `true`, the result of evaluating e_2 is returned, otherwise the result of evaluating e_3 is returned.

A short form of the conditional expression eliminates the else-part. The conditional expression `if ($e_1$) $e_2$` is evaluated as if it was `if ($e_1$) $e_2$ else ()`.

6.17 While Loop Expressions

```
Expr1          ::=  ‘while’ ‘(’ Expr ’)’ {nl} Expr
```

The while loop expression `while ($e_1$) $e_2$` is typed and evaluated as if it was an application of `whileLoop ($e_1$) ($e_2$)` where the hypothetical function `whileLoop` is defined as follows.

```
def whileLoop(cond: => Boolean)(body: => Unit): Unit  =
  if (cond) { body ; whileLoop(cond)(body) } else {}
```

6.18 Do Loop Expressions

```
Expr1          ::=  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ’)’
```

The do loop expression `do $e_1$ while ($e_2$)` is typed and evaluated as if it was the expression `($e_1$ ; while ($e_2$) $e_1$)`. A semicolon preceding the `while` symbol of a do loop expression is ignored.

6.19 For Comprehensions and For Loops

```
Expr1          ::=  ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’)
                       {nl} [‘yield’] Expr
Enumerators    ::=  Generator {semi Enumerator}
Enumerator     ::=  Generator
                 |  Guard
                 |  ‘val’ Pattern1 ‘=’ Expr
Generator      ::=  Pattern1 ‘<-’ Expr [Guard]
Guard          ::=  ‘if’ PostfixExpr
```

A for loop `for ($\mathit{enums}\,$) $e$` executes expression e for each binding generated by the enumerators *enums*. A for comprehension `for ($\mathit{enums}\,$) yield $e$` evaluates expression e for each binding generated by the enumerators *enums* and collects the results. An enumerator sequence always starts with a generator; this can be followed by further generators, value definitions, or guards. A *generator* `$p$ <- $e$` produces bindings from an expression e which is matched in some way against pattern p. A *value definition* `$p$ = $e$` binds the value name p (or several names in a pattern p) to the result of evaluating the expression e. A *guard* `if $e$` contains a boolean expression which restricts enumerated bindings. The precise meaning of generators and guards is defined by translation to invocations of four methods: `map`, `withFilter`, `flatMap`, and `foreach`. These methods can be implemented in different ways for different carrier types.

The translation scheme is as follows. In a first step, every generator `$p$ <- $e$`, where p is not irrefutable for the type of e is replaced by

```
$p$ <- $e$.withFilter { case $p$ => true; case _ => false }
```

Then, the following rules are applied repeatedly until all comprehensions have been eliminated.

- A for comprehension `for ($p$ <- $e\,$) yield $e'$` is translated to `$e$.map { case $p$ => $e'$ }`.
- A for loop `for ($p$ <- $e\,$) $e'$` is translated to `$e$.foreach { case $p$ => $e'$ }`.
- A for comprehension

  ```
  for ($p$ <- $e$; $p'$ <- $e'; \ldots$) yield $e''$
  ```

  where `$\ldots$` is a (possibly empty) sequence of generators, definitions, or guards, is translated to

  ```
  $e$.flatMap { case $p$ => for ($p'$ <- $e'; \ldots$) yield $e''$ }
  ```

- A for loop

  ```
  for ($p$ <- $e$; $p'$ <- $e'; \ldots$) $e''$
  ```

  where `$\ldots$` is a (possibly empty) sequence of generators, definitions, or guards, is translated to

  ```
  $e$.foreach { case $p$ => for ($p'$ <- $e'; \ldots$) $e''$ }
  ```

- A generator `$p$ <- $e$` followed by a guard `if $g$` is translated to a single generator `$p$ <- $e$.withFilter(($x_1 , \ldots , x_n$) => $g\,$)` where $x_1, \ldots, x_n$ are the free variables of p.
- A generator `$p$ <- $e$` followed by a value definition `$p'$ = $e'$` is translated to the following generator of pairs of values, where x and x' are fresh names:

  ```
  ($p$, $p'$) <- for ($x @ p$ <- $e$) yield { val $x' @ p'$ = $e'$; ($x$, $x'$) }
  ```

(63) The following code produces all pairs of numbers between 1 and $n - 1$ whose sums are prime.

```
for  { i <- 1 until n
       j <- 1 until i
       if isPrime(i+j)
} yield (i, j)
```

The for comprehension is translated to:

```
(1 until n)
  .flatMap {
     case i => (1 until i)
       .withFilter { j => isPrime(i+j) }
       .map { case j => (i, j) } }
```

(64) For comprehensions can be used to express vector and matrix algorithms concisely. For instance, here is a function to compute the transpose of a given matrix:

```
def transpose[A](xss: Array[Array[A]]) = {
  for (i <- Array.range(0, xss(0).length)) yield
    for (xs <- xss) yield xs(i)
}
```

Here is a function to compute the scalar product of two vectors:

```
def scalprod(xs: Array[Double], ys: Array[Double]) = {
  var acc = 0.0
  for ((x, y) <- xs zip ys) acc = acc + x * y
  acc
}
```

Finally, here is a function to compute the product of two matrices. Compare with the imperative version of **??**.

```
def matmul(xss: Array[Array[Double]], yss: Array[Array[Double]]) = {
  val ysst = transpose(yss)
  for (xs <- xss) yield
    for (yst <- ysst) yield
      scalprod(xs, yst)
}
```

The code above makes use of the fact that `map`, `flatMap`, `withFilter`, and `foreach` are defined for instances of class `scala.Array`.

6.20 Return Expressions

```
Expr1      ::=  'return' [Expr]
```

A return expression `return $e$` must occur inside the body of some enclosing named method or function. The innermost enclosing named method or function in a source program, f, must have an explicitly declared result type, and the type of e must conform to it.

The return expression evaluates the expression e and returns its value as the result of f. The evaluation of any statements or expressions following the return expression is omitted. The type of a return expression is `scala.Nothing`.

The expression e may be omitted. The return expression `return` is type-checked and evaluated as if it was `return ()`.

An `apply` method which is generated by the compiler as an expansion of an anonymous function does not count as a named function in the source program, and therefore is never the target of a return expression.

Returning from a nested anonymous function is implemented by throwing and catching a `scala.runtime.NonLocalReturnException`. Any exception catches between the point of return and the enclosing methods might see the exception. A key comparison makes sure that these exceptions are only caught by the method instance which is terminated by the return.

If the return expression is itself part of an anonymous function, it is possible that the enclosing instance of f has already returned before the return expression is executed. In that case, the thrown `scala.runtime.NonLocalReturnException` will not be caught, and will propagate up the call stack.

6.21 Throw Expressions

```
Expr1       ::=  'throw' Expr
```

A throw expression `throw $e$` evaluates the expression e. The type of this expression must conform to `Throwable`. If e evaluates to an exception reference, evaluation is aborted with the thrown exception. If e evaluates to `null`, evaluation is instead aborted with a `NullPointerException`. If there is an active `try` expression which handles the thrown exception, evaluation resumes with the handler; otherwise the thread executing the `throw` is aborted. The type of a throw expression is `scala.Nothing`.

6.22 Try Expressions

```
Expr1 ::=  'try' '{' Block '}' ['catch' '{' CaseClauses '}']
           ['finally' Expr]
```

A try expression is of the form `try { $b$ } catch $h$` where the handler h is a pattern matching anonymous function

```
{ case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$ }
```

This expression is evaluated by evaluating the block b. If evaluation of b does not cause an exception to be thrown, the result of b is returned. Otherwise the handler h is applied to the thrown exception.
If the handler contains a case matching the thrown exception, the first such case is invoked. If the handler contains no case matching the thrown exception, the exception is re-thrown.

Let *pt* be the expected type of the try expression. The block b is expected to conform to *pt*. The handler h is expected conform to type `scala.PartialFunction[scala.Throwable, $\mathit{pt}\,$]`. The type of the try expression is the weak least upper bound of the type of b and the result type of h.

A try expression `try { $b$ } finally $e$` evaluates the block b. If evaluation of b does not cause an exception to be thrown, the expression e is evaluated. If an exception is thrown during evaluation of e, the evaluation of the try expression is aborted with the thrown exception. If no exception is thrown during evaluation of e, the result of b is returned as the result of the try expression.

If an exception is thrown during evaluation of b, the finally block e is also evaluated. If another exception e is thrown during evaluation of e, evaluation of the try expression is aborted with the thrown exception. If no exception is thrown during evaluation of e, the original exception thrown in b is re-thrown once evaluation of e has completed. The block b is expected to conform to the expected type of the try expression. The finally expression e is expected to conform to type `Unit`.

A try expression `try { $b$ } catch $e_1$ finally $e_2$` is a shorthand for `try { try { $b$ } catch $e_1$ } finally $e_2$`.

6.23 Anonymous Functions

```
Expr            ::=  (Bindings | [‘implicit’] id | ‘_’) ‘=>’ Expr
ResultExpr    ::= (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block
Bindings        ::=  ‘(’ Binding {‘,’ Binding} ‘)’
Binding         ::=  (id | ‘_’) [‘:’ Type]
```

The anonymous function `($x_1$: $T_1 , \ldots , x_n$: $T_n$) => e` maps parameters x_i of types T_i to a result given by expression e. The scope of each formal parameter x_i is e. Formal parameters must have pairwise distinct names.

If the expected type of the anonymous function is of the form `scala.Function$n$[$S_1 , \ldots , S_n$, $R\,$]`, the expected type of e is R and the type T_i of any of the parameters x_i can be omitted, in which case`$T_i$ = $S_i$` is assumed. If the expected type of the anonymous function is some other type, all formal parameter types must be explicitly given, and the expected type of e is undefined. The type of the anonymous function is`scala.Function$n$[$S_1 , \ldots , S_n$, $T\,$]`, where T is the packed type of e. T must be equivalent to a type which does not refer to any of the formal parameters x_i.

The anonymous function is evaluated as the instance creation expression

```
new scala.Function$n$[$T_1 , \ldots , T_n$, $T$] {
  def apply($x_1$: $T_1 , \ldots , x_n$: $T_n$): $T$ = $e$
}
```

In the case of a single untyped formal parameter, `($x\,$) => $e$` can be abbreviated to `$x$ => $e$`. If an anonymous function `($x$: $T\,$) => $e$` with a single typed parameter appears as the result expression of a block, it can be abbreviated to `$x$: $T$ => e`.

A formal parameter may also be a wildcard represented by an underscore `_`. In that case, a fresh name for the parameter is chosen arbitrarily.

A named parameter of an anonymous function may be optionally preceded by an `implicit` modifier. In that case the parameter is labeled `implicit`; however the parameter section itself does not count as an implicit parameter section in the sense defined here. Hence, arguments to anonymous functions always have to be given explicitly.

(65) Examples of anonymous functions:

```
x => x                             // The identity function

f => g => x => f(g(x))             // Curried function composition

(x: Int,y: Int) => x + y           // A summation function

() => { count += 1; count }        // The function which takes an
                                   // empty parameter list $()$,
                                   // increments a non-local variable
                                   // ‘count’ and returns the new value.

_ => 5                             // The function that ignores its argument
                                   // and always returns 5.
```

6.23.1 Placeholder Syntax for Anonymous Functions

```
SimpleExpr1  ::=  ‘_’
```

An expression (of syntactic category `Expr`) may contain embedded underscore symbols `_` at places where identifiers are legal. Such an expression represents an anonymous function where subsequent occurrences of underscores denote successive parameters.

Define an *underscore section* to be an expression of the form `_:$T$` where T is a type, or else of the form `_`, provided the underscore does not appear as the expression part of a type ascription `_:$T$`.

An expression e of syntactic category `Expr` *binds* an underscore section u, if the following two conditions hold: (1) e properly contains u, and (2) there is no other expression of syntactic category `Expr` which is properly contained in e and which itself properly contains u.

If an expression e binds underscore sections $u_1, \ldots, u_n$, in this order, it is equivalent to the anonymous function `($u'_1$, ... $u'_n$) => $e'$` where each u'_i results from u_i by replacing the underscore with a fresh identifier and e' results from e by replacing each underscore section u_i by u'_i.

(66) The anonymous functions in the left column use placeholder syntax. Each of these is equivalent to the anonymous function on its right.

`_ + 1`	`x => x + 1`
`_ * _`	`(x1, x2) => x1 * x2`
`(_: Int) * 2`	`(x: Int) => (x: Int) * 2`
`if (_) x else y`	`z => if (z) x else y`
`_.map(f)`	`x => x.map(f)`
`_.map(_ + 1)`	`x => x.map(y => y + 1)`

6.24 Constant Expressions

Constant expressions are expressions that the Scala compiler can evaluate to a constant. The definition of “constant expression” depends on the platform, but they include at least the expressions of the following forms:

- A literal of a value class, such as an integer
- A string literal
- A class constructed with `Predef.classOf`
- An element of an enumeration from the underlying platform
- A literal array, of the form `Array$(c_1 , \ldots , c_n)$`, where all of the c_i's are themselves constant expressions
- An identifier defined by a constant value definition.

6.25 Statements

```
BlockStat    ::=  Import
               |  {Annotation} [‘implicit’] Def
               |  {Annotation} {LocalModifier} TmplDef
```

```
               |   Expr1
               |
TemplateStat ::=   Import
               |   {Annotation} {Modifier} Def
               |   {Annotation} {Modifier} Dcl
               |   Expr
               |
```

Statements occur as parts of blocks and templates. A statement can be an import, a definition or an expression, or it can be empty. Statements used in the template of a class definition can also be declarations. An expression that is used as a statement can have an arbitrary value type. An expression statement e is evaluated by evaluating e and discarding the result of the evaluation.

Block statements may be definitions which bind local names in the block. The only modifier allowed in all block-local definitions is `implicit`. When prefixing a class or object definition, modifiers `abstract`, `final`, and `sealed` are also permitted.

Evaluation of a statement sequence entails evaluation of the statements in the order they are written.

6.26 Implicit Conversions

Implicit conversions can be applied to expressions whose type does not match their expected type, to qualifiers in selections, and to unapplied methods. The available implicit conversions are given in the next two sub-sections.

We say, a type T is *compatible* to a type U if T conforms to U after applying eta-expansion and view applications.

6.26.1 Value Conversions

The following five implicit conversions can be applied to an expression e which has some value type T and which is type-checked with some expected type *pt*.

Overloading Resolution
If an expression denotes several possible members of a class, overloading resolution is applied to pick a unique member.

Type Instantiation
An expression e of polymorphic type

```
[$a_1$ >: $L_1$ <: $U_1 , \ldots , a_n$ >: $L_n$ <: $U_n$]$T$
```

which does not appear as the function part of a type application is converted to a type instance of T by determining with local type inference instance types `$T_1 , \ldots , T_n$` for the type variables `$a_1 , \ldots , a_n$` and implicitly embedding e in the type application `$e$[$T_1 , \ldots , T_n$]`.

Numeric Widening
If e has a primitive number type which weakly conforms to the expected type, it is widened to the expected type using one of the numeric conversion methods `toShort`, `toChar`, `toInt`, `toLong`, `toFloat`, `toDouble` defined here.

Numeric Literal Narrowing
If the expected type is `Byte`, `Short` or `Char`, and the expression e is an integer literal fitting in the range of that type, it is converted to the same literal in that type.

Value Discarding
If e has some value type and the expected type is `Unit`, e is converted to the expected type by embedding it in the term `{ $e$; () }`.

View Application
If none of the previous conversions applies, and e's type does not conform to the expected type *pt*, it is attempted to convert e to the expected type with a view.

Dynamic Member Selection
If none of the previous conversions applies, and e is a prefix of a selection $e.x$, and e's type conforms to class `scala.Dynamic`, then the selection is rewritten according to the rules for dynamic member selection.

6.26.2 Method Conversions

The following four implicit conversions can be applied to methods which are not applied to some argument list.

Evaluation
A parameterless method m of type `=> $T$` is always converted to type T by evaluating the expression to which m is bound.

Implicit Application
If the method takes only implicit parameters, implicit arguments are passed following the rules here.

Eta Expansion
Otherwise, if the method is not a constructor, and the expected type *pt* is a function type $(Ts') \Rightarrow T'$, eta-expansion is performed on the expression e.

Empty Application
Otherwise, if e has method type $()T$, it is implicitly applied to the empty argument list, yielding $e()$.

6.26.3 Overloading Resolution

If an identifier or selection e references several members of a class, the context of the reference is used to identify a unique member. The way this is done depends on whether or not e is used as a function. Let $\mathscr{A}$ be the set of members referenced by e.

Assume first that e appears as a function in an application, as in `$e$($e_1 , \ldots , e_m$)`.

One first determines the set of functions that is potentially applicable based on the *shape* of the arguments.

The shape of an argument expression e, written *shape*(e), is a type that is defined as follows:

- For a function expression `($p_1$: $T_1 , \ldots , p_n$: $T_n$) => $b$: (Any $, \ldots ,$ Any) => $\mathit{shape}(b)$`, where `Any` occurs n times in the argument type.
- For a named argument `$n$ = $e$`: *shape*(e).
- For all other expressions: `Nothing`.

Let $\mathscr{B}$ be the set of alternatives in $\mathscr{A}$ that are *applicable* to expressions $(e_1, \ldots, e_n)$ of types $(\mathit{shape}(e_1), \ldots, \mathit{shape}(e_n))$. If there is precisely one alternative in $\mathscr{B}$, that alternative is chosen.

Otherwise, let $S_1, \ldots, S_m$ be the vector of types obtained by typing each argument with an undefined expected type. For every member m in $\mathscr{B}$ one determines whether it is applicable to expressions $(e_1, \ldots, e_m)$ of types $S_1, \ldots, S_m$. It is an error if none of the members in $\mathscr{B}$ is applicable. If there is one single applicable alternative, that alternative is chosen. Otherwise, let $\mathscr{CC}$ be the set of applicable alternatives which don't employ any default argument in the application to $e_1, \ldots, e_m$. It is again an error if $\mathscr{CC}$ is empty. Otherwise, one chooses the *most specific* alternative among the alternatives in $\mathscr{CC}$, according to the following definition of being `as specific as", andmore specific than":

- A parameterized method m of type `($p_1:T_1, \ldots , p_n:T_n$)$U$` is *as specific as* some other member m' of type S if m' is applicable to arguments `($p_1 , \ldots , p_n\,$)` of types $T_1, \ldots, T_n$.
- A polymorphic method of type `[$a_1$ >: $L_1$ <: $U_1 , \ldots , a_n$ >: $L_n$ <: $U_n$]$T$` is as specific as some other member of type S if T is as specific as S under the assumption that for $i = 1, \ldots, n$ each a_i is an abstract type name bounded from below by L_i and from above by U_i.
- A member of any other type is always as specific as a parameterized method or a polymorphic method.
- Given two members of types T and U which are neither parameterized nor polymorphic method types, the member of type T is as specific as the member of type U if the existential dual of T conforms to the existential dual of U. Here, the existential dual of a polymorphic type `[$a_1$ >: $L_1$ <: $U_1 , \ldots ,`

`a_n$ >: $L_n$ <: $U_n$]$T$` is `$T$ forSome { type $a_1$ >: $L_1$ <: $U_1$ $, \ldots ,$ type $a_n$ >: $L_n$ <: $U_n$}`. The existential dual of every other type is the type itself.

The *relative weight* of an alternative A over an alternative B is a number from 0 to 2, defined as the sum of

- 1 if A is as specific as B, 0 otherwise, and
- 1 if A is defined in a class or object which is derived from the class or object defining B, 0 otherwise.

A class or object C is *derived* from a class or object D if one of the following holds:

- C is a subclass of D, or
- C is a companion object of a class derived from D, or
- D is a companion object of a class from which C is derived.

An alternative A is *more specific* than an alternative B if the relative weight of A over B is greater than the relative weight of B over A.

It is an error if there is no alternative in $\mathscr{CC}$ which is more specific than all other alternatives in $\mathscr{CC}$.

Assume next that e appears as a function in a type application, as in `$e$[$\mathit{targs}\,$]`. Then all alternatives in $\mathscr{A}$ which take the same number of type parameters as there are type arguments in *targs* are chosen. It is an error if no such alternative exists. If there are several such alternatives, overloading resolution is applied again to the whole expression `$e$[$\mathit{targs}\,$]`.

Assume finally that e does not appear as a function in either an application or a type application. If an expected type is given, let $\mathscr{B}$ be the set of those alternatives in $\mathscr{A}$ which are compatible to it. Otherwise, let $\mathscr{B}$ be the same as $\mathscr{A}$. We choose in this case the most specific alternative among all alternatives in $\mathscr{B}$. It is an error if there is no alternative in $\mathscr{B}$ which is more specific than all other alternatives in $\mathscr{B}$.

(67) Consider the following definitions:

```
class A extends B {}
def f(x: B, y: B) = $\ldots$
def f(x: A, y: B) = $\ldots$
val a: A
val b: B
```

Then the application `f(b, b)` refers to the first definition of f whereas the application `f(a, a)` refers to the second. Assume now we add a third overloaded definition

```
def f(x: B, y: A) = $\ldots$
```

Then the application `f(a, a)` is rejected for being ambiguous, since no most specific applicable signature exists.

6.26.4 Local Type Inference

Local type inference infers type arguments to be passed to expressions of polymorphic type. Say e is of type $[a_1$ >: L_1 <: $U_1, \ldots, a_n$ >: L_n <: $U_n]T$ and no explicit type parameters are given.

Local type inference converts this expression to a type application `$e$[$T_1 , \ldots , T_n$]`. The choice of the type arguments $T_1, \ldots, T_n$ depends on the context in which the expression appears and on the expected type *pt*. There are three cases.

Case 1: Selections
If the expression appears as the prefix of a selection with a name x, then type inference is *deferred* to the whole expression $e.x$. That is, if $e.x$ has type S, it is now treated as having type $[a_1$ >: L_1 <: $U_1, \ldots, a_n$ >: L_n <: $U_n]S$, and local type inference is applied in turn to infer type arguments for $a_1, \ldots, a_n$, using the context in which $e.x$ appears.

Case 2: Values
If the expression e appears as a value without being applied to value arguments, the type arguments are inferred by solving a constraint system which relates the expression's type T with the expected type *pt*. Without loss of generality we can assume that T is a value type; if it is a method type we apply eta-expansion to convert it to a function type. Solving means finding a substitution σ of types T_i for the type parameters a_i such that

- None of inferred types T_i is a singleton type
- All type parameter bounds are respected, i.e. σL_i <: σa_i and σa_i <: σU_i for $i = 1, \ldots, n$.
- The expression's type conforms to the expected type, i.e. σT <: σ*pt*.

It is a compile time error if no such substitution exists.
If several substitutions exist, local-type inference will choose for each type variable a_i a minimal or maximal type T_i of the solution space. A *maximal* type T_i will be chosen if the type parameter a_i appears contravariantly in the type T of the expression. A *minimal* type T_i will be chosen in all other situations, i.e. if the variable appears covariantly, non-variantly or not at all in the type T. We call such a substitution an *optimal solution* of the given constraint system for the type T.

Case 3: Methods
The last case applies if the expression e appears in an application $e(d_1, \ldots, d_m)$. In that case T is a method type $(p_1 : R_1, \ldots, p_m : R_m)T'$. Without loss of generality we can assume that the result type T' is a value type; if it is a method type we apply eta-expansion to convert it to a function type. One computes first the types S_j of the argument expressions d_j, using two alternative schemes. Each argument expression d_j is typed first

with the expected type R_j, in which the type parameters $a_1, \ldots, a_n$ are taken as type constants. If this fails, the argument d_j is typed instead with an expected type R'_j which results from R_j by replacing every type parameter in $a_1, \ldots, a_n$ with {*undefined*}.

In a second step, type arguments are inferred by solving a constraint system which relates the method's type with the expected type pt and the argument types $S_1, \ldots, S_m$. Solving the constraint system means finding a substitution σ of types T_i for the type parameters a_i such that

- *None of inferred types T_i is a singleton type*
- *All type parameter bounds are respected, i.e. σL_i <: σa_i and σa_i <: σU_i for $i = 1, \ldots, n$.*
- *The method's result type T' conforms to the expected type, i.e. $\sigma T'$ <: σ pt.*
- *Each argument type weakly conforms to the corresponding formal parameter type, i.e. $\sigma S_j <:_w \sigma R_j$ for $j = 1, \ldots, m$.*

It is a compile time error if no such substitution exists. If several solutions exist, an optimal one for the type T' is chosen.

All or parts of an expected type pt may be undefined. The rules for conformance are extended to this case by adding the rule that for any type T the following two statements are always true: undefined <: T and T <: undefined

It is possible that no minimal or maximal solution for a type variable exists, in which case a compile-time error results. Because <: is a pre-order, it is also possible that a solution set has several optimal solutions for a type. In that case, a Scala compiler is free to pick any one of them.

(68) Consider the two methods:

```
def cons[A](x: A, xs: List[A]): List[A] = x :: xs
def nil[B]: List[B] = Nil
```

and the definition

```
val xs = cons(1, nil)
```

The application of `cons` is typed with an undefined expected type. This application is completed by local type inference to `cons[Int](1, nil)`. Here, one uses the following reasoning to infer the type argument `Int` for the type parameter `a`:

First, the argument expressions are typed. The first argument `1` has type `Int` whereas the second argument `nil` is itself polymorphic. One tries to type-check `nil` with an expected type `List[a]`. This leads to the constraint system

```
List[b?] <: List[a]
```

where we have labeled `b?` with a question mark to indicate that it is a variable in the constraint system. Because class `List` is covariant, the optimal solution of this constraint is

```
b = scala.Nothing
```

In a second step, one solves the following constraint system for the type parameter `a` of `cons`:

```
Int <: a?
List[scala.Nothing] <: List[a?]
List[a?] <: $\mbox{\sl undefined}$
```

The optimal solution of this constraint system is

```
a = Int
```

so `Int` is the type inferred for `a`.

(69) *Consider now the definition*

```
val ys = cons("abc", xs)
```

where `xs` is defined of type `List[Int]` as before. In this case local type inference proceeds as follows.

First, the argument expressions are typed. The first argument `"abc"` has type `String`. The second argument `xs` is first tried to be typed with expected type `List[a]`. This fails, as `List[Int]` is not a subtype of `List[a]`. Therefore, the second strategy is tried; `xs` is now typed with expected type `List[$\mbox{\sl undefined}$]`. This succeeds and yields the argument type `List[Int]`.

In a second step, one solves the following constraint system for the type parameter `a` of `cons`:

```
String <: a?
List[Int] <: List[a?]
List[a?] <: $\mbox{\sl undefined}$
```

The optimal solution of this constraint system is

```
a = scala.Any
```

so `scala.Any` is the type inferred for `a`.

6.26.5 Eta Expansion

Eta-expansion *converts an expression of method type to an equivalent expression of function type. It proceeds in two steps.*

First, one identifes the maximal sub-expressions of e; let's say these are $e_1, \ldots, e_m$. For each of these, one creates a fresh name x_i. Let e' be the expression resulting from replacing every maximal subexpression e_i in e by the corresponding fresh name x_i. Second, one creates a fresh name y_i for every argument type T_i of the method ($i = 1, \ldots, n$). The result of eta-conversion is then:

```
{ val $x_1$ = $e_1$;
  $\ldots$
  val $x_m$ = $e_m$;
  ($y_1: T_1 , \ldots , y_n: T_n$) => $e'$($y_1 , \ldots , y_n$)
}
```

6.26.6 Dynamic Member Selection

The standard Scala library defines a trait `scala.Dynamic` *which defines a member @invoke-Dynamic@ as follows:*

```
package scala
trait Dynamic {
  def applyDynamic (name: String, args: Any*): Any
  ...
}
```

Assume a selection of the form $e.x$ where the type of e conforms to `scala.Dynamic`*. Further assuming the selection is not followed by any function arguments, such an expression can be rewitten under the conditions given here to:*

```
$e$.applyDynamic("$x$")
```

If the selection is followed by some arguments, e.g. $e.x(\mathit{args})$, then that expression is rewritten to

```
$e$.applyDynamic("$x$", $\mathit{args}$)
```

Chapter 7

Implicit Parameters and Views

7.1 The Implicit Modifier

```
LocalModifier  ::= ‘implicit’
ParamClauses   ::= {ParamClause} [nl] ‘(’ ‘implicit’ Params ‘)’
```

*Template members and parameters labeled with an `implicit` modifier can be passed to implicit parameters and can be used as implicit conversions called views. The **`implicit`** modifier is illegal for all type members, as well as for top-level objects.*

(70) *The following code defines an abstract class of monoids and two concrete implementations, `StringMonoid` and `IntMonoid`. The two implementations are marked implicit.*

```
abstract class Monoid[A] extends SemiGroup[A] {
  def unit: A
  def add(x: A, y: A): A
}
object Monoids {
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid extends Monoid[Int] {
    def add(x: Int, y: Int): Int = x + y
    def unit: Int = 0
  }
}
```

7.2 Implicit Parameters

An implicit parameter list (`implicit $p_1$,$\ldots$,$p_n$`) of a method marks the parameters $p_1, \ldots, p_n$ as implicit. A method or constructor can have only one implicit parameter list, and it must be the last parameter list given.

A method with implicit parameters can be applied to arguments just like a normal method. In this case the `implicit` label has no effect. However, if such a method misses arguments for its implicit parameters, such arguments will be automatically provided.

*The actual arguments that are eligible to be passed to an implicit parameter of type T fall into two categories. First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition or an implicit parameter. An eligible identifier may thus be a local name, or a member of an enclosing template, or it may be have been made accessible without a prefix through an import clause. If there are no eligible identifiers under this rule, then, second, eligible are also all **`implicit`** members of some object that belongs to the implicit scope of the implicit parameter's type, T.*

The implicit scope *of a type T consists of all companion modules of classes that are associated with the implicit parameter's type. Here, we say a class C is* associated *with a type T, if it is a base class of some part of T. The* parts *of a type T are:*

- *if T is a compound type `$T_1$ with $\ldots$ with $T_n$`, the union of the parts of $T_1, \ldots, T_n$, as well as T itself,*
- *if T is a parameterized type `$S$[$T_1 , \ldots , T_n$]`, the union of the parts of S and $T_1, \ldots, T_n$,*
- *if T is a singleton type `$p$.type`, the parts of the type of p,*
- *if T is a type projection `$S$#$U$`, the parts of S as well as T itself,*
- *in all other cases, just T itself.*

If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of static overloading resolution. If the parameter has a default argument and no implicit argument can be found the default argument is used.

(71) *Assuming the classes from **??**, here is a method which computes the sum of a list of elements using the monoid's* add *and* `unit` *operations.*

```
def sum[A](xs: List[A])(implicit m: Monoid[A]): A =
  if (xs.isEmpty) m.unit
  else m.add(xs.head, sum(xs.tail))
```

The monoid in question is marked as an implicit parameter, and can therefore be inferred based on the type of the list. Consider for instance the call `sum(List(1, 2, 3))` in a context where `stringMonoid` and `intMonoid` are visible. We know that the formal type parameter `a` *of* `sum` *needs to be instantiated to* `Int`*. The only eligible object which matches the implicit formal parameter type* `Monoid[Int]` *is* `intMonoid` *so this object will be passed as implicit parameter.*

This discussion also shows that implicit parameters are inferred after any type arguments are inferred.

Implicit methods can themselves have implicit parameters. An example is the following method from module `scala.List`*, which injects lists into the* `scala.Ordered` *class, provided the element type of the list is also convertible to this type.*

```
implicit def list2ordered[A](x: List[A])
  (implicit elem2ordered: A => Ordered[A]): Ordered[List[A]] =
  ...
```

Assume in addition a method

```
implicit def int2ordered(x: Int): Ordered[Int]
```

that injects integers into the `Ordered` *class. We can now define a* `sort` *method over ordered lists:*

```
def sort[A](xs: List[A])(implicit a2ordered: A => Ordered[A]) = ...
```

We can apply `sort` *to a list of lists of integers* `yss: List[List[Int]]` *as follows:*

```
sort(yss)
```

The call above will be completed by passing two nested implicit arguments:

```
sort(yss)(xs: List[Int] => list2ordered[Int](xs)(int2ordered)) .
```

The possibility of passing implicit arguments to implicit arguments raises the possibility of an infinite recursion. For instance, one might try to define the following method, which injects every *type into the* `Ordered` *class:*

```
implicit def magic[A](x: A)(implicit a2ordered: A => Ordered[A]): Ordered[A] =
  a2ordered(x)
```

Now, if one tried to apply `sort` *to an argument* `arg` *of a type that did not have another injection into the* `Ordered` *class, one would obtain an infinite expansion:*

```
sort(arg)(x => magic(x)(x => magic(x)(x => ... )))
```

To prevent such infinite expansions, the compiler keeps track of a stack of "open implicit types" for which implicit arguments are currently being searched. Whenever an implicit argument for type T is searched, the "core type" of T is added to the stack. Here, the core type *of T is T with aliases expanded, top-level type annotations and refinements removed, and occurrences of top-level existentially bound variables replaced by their upper bounds. The core type is removed from the stack once the search for the implicit argument either definitely fails or succeeds. Everytime a core type is added to the stack, it is checked that this type does not dominate any of the other types in the set.*

Here, a core type T dominates *a type U if T is equivalent to U, or if the top-level type constructors of T and U have a common element and T is more complex than U.*

The set of top-level type constructors *$\mathit{ttcs}(T)$ of a type T depends on the form of the type:*

> *For a type designator,*
> $\mathit{ttcs}(p.c) \;=\; \{c\}$;
> *For a parameterized type,*
> $\mathit{ttcs}(p.c[\mathit{targs}]) \;=\; \{c\}$;
> *For a singleton type,*
> $\mathit{ttcs}(p.type) \;=\; \mathit{ttcs}(T)$, *provided p has type T;*
> *For a compound type,*
> `$\mathit{ttcs}(T_1$ with $\ldots$ with $T_n)$` $= \mathit{ttcs}(T_1) \cup \ldots \cup \mathit{ttcs}(T_n)$.

The complexity *$\mathit{complexity}(T)$ of a core type is an integer which also depends on the form of the type:*

> *For a type designator,*
> $\mathit{complexity}(p.c) \;=\; 1 + \mathit{complexity}(p)$
> *For a parameterized type,*
> $\mathit{complexity}(p.c[\mathit{targs}]) \;=\; 1 + \Sigma\mathit{complexity}(\mathit{targs})$
> *For a singleton type denoting a package p,*
> $\mathit{complexity}(p.type) \;=\; 0$
> *For any other singleton type,*
> $\mathit{complexity}(p.type) \;=\; 1 + \mathit{complexity}(T)$, *provided p has type T;*
> *For a compound type,*
> `$\mathit{complexity}(T_1$ with $\ldots$ with $T_n)$` $=$ $\Sigma\mathit{complexity}(T_i)$

(72) *When typing `sort(xs)` for some list `xs` of type `List[List[List[Int]]]`, the sequence of types for which implicit arguments are searched is*

```
List[List[Int]] => Ordered[List[List[Int]]],
List[Int] => Ordered[List[Int]]
Int => Ordered[Int]
```

All types share the common type constructor `scala.Function1`*, but the complexity of the each new type is lower than the complexity of the previous types. Hence, the code typechecks.*

(73) *Let* `ys` *be a list of some type which cannot be converted to* `Ordered`*. For instance:*

```
val ys = List(new IllegalArgumentException, new ClassCastException, new Error)
```

Assume that the definition of `magic` *above is in scope. Then the sequence of types for which implicit arguments are searched is*

```
Throwable => Ordered[Throwable],
Throwable => Ordered[Throwable],
...
```

Since the second type in the sequence is equal to the first, the compiler will issue an error signalling a divergent implicit expansion.

7.3 Views

Implicit parameters and methods can also define implicit conversions called views. A view *from type S to type T is defined by an implicit value which has function type* `$S$=>$T$` *or* `(=>$S$)=>$T$` *or by a method convertible to a value of that type.*

Views are applied in three situations.

1. *If an expression e is of type T, and T does not conform to the expression's expected type pt. In this case an implicit v is searched which is applicable to e and whose result type conforms to pt. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of* `$T$ => $\mathit{pt}$`*. If such a view is found, the expression e is converted to* `$v$($e$)`*.*
2. *In a selection $e.m$ with e of type T, if the selector m does not denote a member of T. In this case, a view v is searched which is applicable to e and whose result contains a member named m. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the selection $e.m$ is converted to* `$v$($e$).$m$`*.*
3. *In a selection $e.m(\mathit{args})$ with e of type T, if the selector m denotes some member(s) of T, but none of these members is applicable to the arguments args. In this case a view v is searched which is applicable to e and whose result contains a method m which is applicable to args. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the selection $e.m$ is converted to* `$v$($e$).$m(\mathit{args})$`*.*

The implicit view, if it is found, can accept is argument e as a call-by-value or as a call-by-name parameter. However, call-by-value implicits take precedence over call-by-name implicits.

As for implicit parameters, overloading resolution is applied if there are several possible candidates (of either the call-by-value or the call-by-name category).

(74) *Class* `scala.Ordered[A]` *contains a method*

```
def <= [B >: A](that: B)(implicit b2ordered: B => Ordered[B]): Boolean .
```

Assume two lists `xs` *and* `ys` *of type* `List[Int]` *and assume that the* `list2ordered` *and* `int2ordered` *methods defined here are in scope. Then the operation*

```
xs <= ys
```

is legal, and is expanded to:

```
list2ordered(xs)(int2ordered).<=
  (ys)
  (xs => list2ordered(xs)(int2ordered))
```

The first application of `list2ordered` *converts the list* `xs` *to an instance of class* `Ordered`*, whereas the second occurrence is part of an implicit parameter passed to the* `<=` *method.*

7.4 Context Bounds and View Bounds

```
TypeParam ::= (id | '_') [TypeParamClause] ['>:' Type] ['<:' Type]
              {'<%' Type} {':' Type}
```

A type parameter A of a method or non-trait class may have one or more view bounds A <% T. In this case the type parameter may be instantiated to any type S which is convertible by application of a view to the bound T.

A type parameter A of a method or non-trait class may also have one or more context bounds A : T. In this case the type parameter may be instantiated to any type S for which evidence *exists at the instantiation point that S satisfies the bound T. Such evidence consists of an implicit value with type $T[S]$.*

A method or class containing type parameters with view or context bounds is treated as being equivalent to a method with implicit parameters. Consider first the case of a single parameter with view and/or context bounds such as:

```
def $f$[$A$ <% $T_1$ ... <% $T_m$ : $U_1$ : $U_n$]($\mathit{ps}$): $R$ = ...
```

Then the method definition above is expanded to

```
def $f$[$A$]($\mathit{ps}$)(implicit $v_1$: $A$ => $T_1$, ..., $v_m$: $A$ => $T_m$,
                             $w_1$: $U_1$[$A$], ..., $w_n$: $U_n$[$A$]): $R$ = ...
```

where the v_i *and* w_j *are fresh names for the newly introduced implicit parameters. These parameters are called* evidence parameters.

If a class or method has several view- or context-bounded type parameters, each such type parameter is expanded into evidence parameters in the order they appear and all the resulting evidence parameters are concatenated in one implicit parameter section. Since traits do not take constructor parameters, this translation does not work for them. Consequently, type-parameters in traits may not be view- or context-bounded. Also, a method or class with view- or context bounds may not define any additional implicit parameters.

(75) *The <= method mentioned in* **??** *can be declared more concisely as follows:*

```
def <= [B >: A <% Ordered[B]](that: B): Boolean
```

7.5 Manifests

Manifests are type descriptors that can be automatically generated by the Scala compiler as arguments to implicit parameters. The Scala standard library contains a hierarchy of four manifest classes, with `OptManifest` *at the top. Their signatures follow the outline below.*

```
trait OptManifest[+T]
object NoManifest extends OptManifest[Nothing]
trait ClassManifest[T] extends OptManifest[T]
trait Manifest[T] extends ClassManifest[T]
```

If an implicit parameter of a method or constructor is of a subtype $M[T]$ *of class* `OptManifest[T]`, a manifest is determined for $M[S]$, *according to the following rules.*

First if there is already an implicit argument that matches $M[T]$, *this argument is selected.*

Otherwise, let Mobj be the companion object `scala.reflect.Manifest` *if* M *is trait* `Manifest`, *or be the companion object* `scala.reflect.ClassManifest` *otherwise. Let* M' *be the trait* `Manifest` *if* M *is trait* `Manifest`, *or be the trait* `OptManifest` *otherwise.*
Then the following rules apply.

1. *If* T *is a value class or one of the classes* `Any`, `AnyVal`, `Object`, `Null`, *or* `Nothing`, *a manifest for it is generated by selecting the corresponding manifest value* `Manifest.$T$`, *which exists in the* `Manifest` *module.*

2. *If T is an instance of `Array[$S$]`, a manifest is generated with the invocation `$\mathit{Mobj}$.arrayType[S](m)`, where m is the manifest determined for $M[S]$.*
3. *If T is some other class type $S\#C[U_1, \ldots, U_n]$ where the prefix type S cannot be statically determined from the class C, a manifest is generated with the invocation `$\mathit{Mobj}$.classType[T]($m_0$, classOf[T], $ms$)` where m_0 is the manifest determined for $M'[S]$ and ms are the manifests determined for $M'[U_1], \ldots, M'[U_n]$.*
4. *If T is some other class type with type arguments $U_1, \ldots, U_n$, a manifest is generated with the invocation `$\mathit{Mobj}$.classType[T](classOf[T], $ms$)` where ms are the manifests determined for $M'[U_1], \ldots, M'[U_n]$.*
5. *If T is a singleton type `$p$.type`, a manifest is generated with the invocation `$\mathit{Mobj}$.singleType[T]($p$)`*
6. *If T is a refined type $T'\{R\}$, a manifest is generated for T'. (That is, refinements are never reflected in manifests).*
7. *If T is an intersection type `$T_1$ with $, \ldots ,$ with $T_n$` where $n > 1$, the result depends on whether a full manifest is to be determined or not. If M is trait `Manifest`, then a manifest is generated with the invocation `Manifest.intersectionType[T]($ms$)` where ms are the manifests determined for $M[T_1], \ldots, M[T_n]$. Otherwise, if M is trait `ClassManifest`, then a manifest is generated for the intersection dominator of the types $T_1, \ldots, T_n$.*
8. *If T is some other type, then if M is trait `OptManifest`, a manifest is generated from the designator `scala.reflect.NoManifest`. If M is a type different from `OptManifest`, a static error results.*

Chapter 8

Pattern Matching

8.1 Patterns

```
Pattern        ::=  Pattern1 { ‘|’ Pattern1 }
Pattern1       ::=  varid ‘:’ TypePat
                 |  ‘_’ ‘:’ TypePat
                 |  Pattern2
Pattern2       ::=  varid [‘@’ Pattern3]
                 |  Pattern3
Pattern3       ::=  SimplePattern
                 |  SimplePattern {id [nl] SimplePattern}
SimplePattern  ::=  ‘_’
                 |  varid
                 |  Literal
                 |  StableId
                 |  StableId ‘(’ [Patterns] ‘)’
                 |  StableId ‘(’ [Patterns ‘,’] [varid ‘@’] ‘_’ ‘*’ ‘)’
                 |  ‘(’ [Patterns] ‘)’
                 |  XmlPattern
Patterns       ::=  Pattern {‘,’ Patterns}
```

A pattern is built from constants, constructors, variables and type tests. Pattern matching tests whether a given value (or sequence of values) has the shape defined by a pattern, and, if it does, binds the variables in the pattern to the corresponding components of the value (or sequence of values). The same variable name may not be bound more than once in a pattern.

(76) Some examples of patterns are:

(a) *The pattern* `ex: IOException` *matches all instances of class* `IOException`*, binding variable* `ex` *to the instance.*
(b) *The pattern* `Some(x)` *matches values of the form* `Some($v$)`*, binding* `x` *to the argument value v of the* `Some` *constructor.*
(c) *The pattern* `(x, _)` *matches pairs of values, binding* `x` *to the first component of the pair. The second component is matched with a wildcard pattern.*
(d) *The pattern* `x :: y :: xs` *matches lists of length ≥ 2, binding* `x` *to the list's first element,* `y` *to the list's second element, and* `xs` *to the remainder.*
(e) *The pattern* `1 | 2 | 3` *matches the integers between 1 and 3.*

Pattern matching is always done in a context which supplies an expected type of the pattern. We distinguish the following kinds of patterns.

8.1.1 Variable Patterns

```
SimplePattern   ::=  '_'
                  |  varid
```

A variable pattern x is a simple identifier which starts with a lower case letter. It matches any value, and binds the variable name to that value. The type of x is the expected type of the pattern as given from outside. A special case is the wild-card pattern _ *which is treated as if it was a fresh variable on each occurrence.*

8.1.2 Typed Patterns

```
Pattern1        ::=  varid ':' TypePat
                  |  '_' ':' TypePat
```

A typed pattern $x : T$ consists of a pattern variable x and a type pattern T. The type of x is the type pattern T, where each type variable and wildcard is replaced by a fresh, unknown type. This pattern matches any value matched by the type pattern T; it binds the variable name to that value.

8.1.3 Pattern Binders

```
Pattern2        ::=  varid '@' Pattern3
```

A pattern binder `$x$@$p$` *consists of a pattern variable x and a pattern p. The type of the variable x is the static type T of the pattern p. This pattern matches any value v matched by the pattern p, provided the run-time type of v is also an instance of T, and it binds the variable name to that value.*

8.1.4 Literal Patterns

```
SimplePattern   ::=  Literal
```

A literal pattern L matches any value that is equal (in terms of $==$) to the literal L. The type of L must conform to the expected type of the pattern.

8.1.5 Stable Identifier Patterns

```
SimplePattern   ::=  StableId
```

A stable identifier pattern is a stable identifier r. The type of r must conform to the expected type of the pattern. The pattern matches any value v such that `$r$ == $v$` (see here).

To resolve the syntactic overlap with a variable pattern, a stable identifier pattern may not be a simple name starting with a lower-case letter. However, it is possible to enclose a such a variable name in backquotes; then it is treated as a stable identifier pattern.

(77) Consider the following function definition:

```
def f(x: Int, y: Int) = x match {
  case y => ...
}
```

Here, `y` is a variable pattern, which matches any value. If we wanted to turn the pattern into a stable identifier pattern, this can be achieved as follows:

```
def f(x: Int, y: Int) = x match {
  case `y` => ...
}
```

Now, the pattern matches the `y` parameter of the enclosing function `f`. That is, the match succeeds only if the `x` argument and the `y` argument of `f` are equal.

8.1.6 Constructor Patterns

```
SimplePattern   ::=  StableId '(' [Patterns] ')
```

A constructor pattern is of the form $c(p_1, \ldots, p_n)$ where $n \geq 0$. It consists of a stable identifier c, followed by element patterns $p_1, \ldots, p_n$. The constructor c is a simple or qualified name which denotes a case class. If the case class is monomorphic, then it must conform to the expected type of the pattern, and the formal parameter types of x's primary constructor are taken as the expected types of the element patterns $p_1, \ldots, p_n$. If the case class is polymorphic, then its type parameters are instantiated so that the instantiation of c conforms *to the expected type of the pattern. The instantiated formal parameter types of c's primary constructor are then taken as the expected types of the component patterns $p_1, \ldots, p_n$. The pattern matches all objects created from constructor invocations $c(v_1, \ldots, v_n)$ where each element pattern p_i matches the corresponding value v_i.*

A special case arises when c's formal parameter types end in a repeated parameter. This is further discussed here.

8.1.7 Tuple Patterns

```
SimplePattern   ::=  '(' [Patterns] ')'
```

A tuple pattern `($p_1 , \ldots , p_n$)` is an alias for the constructor pattern `scala.Tuple$n$($p_1 , \ldots , p_n$)`, where $n \geq 2$. The empty tuple `()` is the unique value of type `scala.Unit`.

8.1.8 Extractor Patterns

```
SimplePattern   ::=  StableId '(' [Patterns] ')'
```

An extractor pattern $x(p_1, \ldots, p_n)$ where $n \geq 0$ is of the same syntactic form as a constructor pattern. However, instead of a case class, the stable identifier x denotes an object which has a member method named `unapply` or `unapplySeq` that matches the pattern.

An `unapply` method in an object x matches *the pattern $x(p_1, \ldots, p_n)$ if it takes exactly one argument and one of the following applies:*

- *$n = 0$ and `unapply`'s result type is `Boolean`. In this case the extractor pattern matches all values v for which `$x$.unapply($v$)` yields `true`.*
- *$n = 1$ and `unapply`'s result type is `Option[$T$]`, for some type T. In this case, the (only) argument pattern p_1 is typed in turn with expected type T. The extractor pattern matches then all values v for which `$x$.unapply($v$)` yields a value of form `Some($v_1$)`, and p_1 matches v_1.*
- *$n > 1$ and `unapply`'s result type is `Option[($T_1 , \ldots , T_n$)]`, for some types $T_1, \ldots, T_n$. In this case, the argument patterns $p_1, \ldots, p_n$ are typed in turn with expected types $T_1, \ldots, T_n$. The extractor pattern matches then all values v for which `$x$.unapply($v$)` yields a value of form `Some(($v_1 , \ldots , v_n$))`, and each pattern p_i matches the corresponding value v_i.*

An `unapplySeq` method in an object x matches the pattern $x(p_1, \ldots, p_n)$ if it takes exactly one argument and its result type is of the form `Option[$S$]`, where S is a subtype of `Seq[$T$]` for some element type T. This case is further discussed here.

(78) *The `Predef` object contains a definition of an extractor object `Pair`:*

```
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}
```

This means that the name `Pair` can be used in place of `Tuple2` for tuple formation as well as for deconstruction of tuples in patterns. Hence, the following is possible:

```
val x = (1, 2)
val y = x match {
  case Pair(i, s) => Pair(s + i, i * i)
}
```

8.1.9 Pattern Sequences

```
SimplePattern ::= StableId '(' [Patterns ','] [varid '@'] '_' '*' ')'
```

A pattern sequence $p_1, \ldots, p_n$ appears in two contexts. First, in a constructor pattern $c(q_1, \ldots, q_m, p_1, \ldots, p_n)$, where c is a case class which has $m + 1$ primary constructor parameters, ending in a repeated parameter of type $S*$. Second, in an extractor pattern $x(p_1, \ldots, p_n)$ if the extractor object x has an `unapplySeq` method with a result type conforming to `Seq[$S$]`, but does not have an `unapply` method that matches $p_1, \ldots, p_n$. The expected type for the pattern sequence is in each case the type S.

The last pattern in a pattern sequence may be a sequence wildcard `_*`. *Each element pattern p_i is type-checked with S as expected type, unless it is a sequence wildcard. If a final sequence wildcard is present, the pattern matches all values v that are sequences which start with elements matching patterns $p_1, \ldots, p_{n-1}$. If no final sequence wildcard is given, the pattern matches all values v that are sequences of length n which consist of elements matching patterns $p_1, \ldots, p_n$.*

8.1.10 Infix Operation Patterns

```
Pattern3  ::=  SimplePattern {id [nl] SimplePattern}
```

An infix operation pattern $p; op; q$ is a shorthand for the constructor or extractor pattern $op(p, q)$. The precedence and associativity of operators in patterns is the same as in expressions.

An infix operation pattern $p; op; (q_1, \ldots, q_n)$ is a shorthand for the constructor or extractor pattern $op(p, q_1, \ldots, q_n)$.

8.1.11 Pattern Alternatives

```
Pattern   ::=  Pattern1 { '|' Pattern1 }
```

A pattern alternative `$p_1$ | $\ldots$ | $p_n$` consists of a number of alternative patterns p_i. All alternative patterns are type checked with the expected type of the pattern. They may no bind variables other than wildcards. The alternative pattern matches a value v if at least one its alternatives matches v.

8.1.12 XML Patterns

XML patterns are treated here.

8.1.13 Regular Expression Patterns

Regular expression patterns have been discontinued in Scala from version 2.0.

Later version of Scala provide a much simplified version of regular expression patterns that cover most scenarios of non-text sequence processing. A sequence pattern *is a pattern that stands in a position where either (1) a pattern of a type `T` which is conforming to `Seq[A]` for some `A` is expected, or (2) a case class constructor that has an iterated formal parameter `A*`. A wildcard star pattern `_*` in the rightmost position stands for arbitrary long sequences. It can be bound to variables using `@`, as usual, in which case the variable will have the type `Seq[A]`.*

8.1.14 Irrefutable Patterns

A pattern p is irrefutable *for a type T, if one of the following applies:*

1. *p is a variable pattern,*
2. *p is a typed pattern $x : T'$, and $T <: T'$,*
3. *p is a constructor pattern $c(p_1, \ldots, p_n)$, the type T is an instance of class c, the primary constructor of type T has argument types $T_1, \ldots, T_n$, and each p_i is irrefutable for T_i.*

8.2 Type Patterns

```
TypePat           ::=  Type
```

Type patterns consist of types, type variables, and wildcards. A type pattern T is of one of the following forms:

- *A reference to a class C, $p.C$, or `$T$#$C$`. This type pattern matches any non-null instance of the given class. Note that the prefix of the class, if it is given, is relevant for determining class instances. For instance, the pattern $p.C$ matches only instances of classes C which were created with the path p as prefix.*

The bottom types `scala.Nothing` and `scala.Null` cannot be used as type patterns, because they would match nothing in any case.

- *A singleton type `$p$.type`. This type pattern matches only the value denoted by the path p (that is, a pattern match involved a comparison of the matched value with p using method `eq` in class `AnyRef`).*
- *A compound type pattern `$T_1$ with $\ldots$ with $T_n$` where each T_i is a type pattern. This type pattern matches all values that are matched by each of the type patterns T_i.*
- *A parameterized type pattern $T[a_1, \ldots, a_n]$, where the a_i are type variable patterns or wildcards `_`. This type pattern matches all values which match T for some arbitrary instantiation of the type variables and wildcards. The bounds or alias type of these type variable are determined as described here.*
- *A parameterized type pattern `scala.Array$[T_1]$`, where T_1 is a type pattern. This type pattern matches any non-null instance of type `scala.Array$[U_1]$`, where U_1 is a type matched by T_1.*

Types which are not of one of the forms described above are also accepted as type patterns. However, such type patterns will be translated to their erasure. The Scala compiler will issue an "unchecked" warning for these patterns to flag the possible loss of type-safety.

A type variable pattern *is a simple identifier which starts with a lower case letter. However, the predefined primitive type aliases `unit`, `boolean`, `byte`, `short`, `char`, `int`, `long`, `float`, and `double` are not classified as type variable patterns.*

8.3 Type Parameter Inference in Patterns

Type parameter inference is the process of finding bounds for the bound type variables in a typed pattern or constructor pattern. Inference takes into account the expected type of the pattern.

Type parameter inference for typed patterns.
Assume a typed pattern $p : T'$. Let T result from T' where all wildcards in T' are renamed to fresh variable names. Let $a_1, \ldots, a_n$ be the type variables in T. These type variables are considered bound in the pattern. Let the expected type of the pattern be pt.

Type parameter inference constructs first a set of subtype constraints over the type variables a_i. The initial constraints set $\mathcal{C}_0$ reflects just the bounds of these type variables. That is, *assuming T has bound type variables $a_1, \ldots, a_n$ which correspond to class type parameters $a'_1, \ldots, a'_n$ with lower bounds $L_1, \ldots, L_n$ and upper bounds $U_1, \ldots, U_n$, $\mathcal{C}_0$ contains the constraints*

$$
\begin{array}{llll}
a_i & <: & \sigma U_i & (i = 1, \ldots, n) \\
\sigma L_i & <: & a_i & (i = 1, \ldots, n)
\end{array}
$$

where σ is the substitution $[a'_1 := a_1, \ldots, a'_n := a_n]$.

The set $\mathcal{C}_0$ is then augmented by further subtype constraints. There are two cases.

Case 1:
If there exists a substitution σ over the type variables $a_i, \ldots, a_n$ such that σT conforms to pt, one determines the weakest subtype constraints $\mathcal{C}_1$ over the type variables $a_1, \ldots, a_n$ such that $\mathcal{C}_0 \wedge \mathcal{C}_1$ implies that T conforms to pt.

Case 2:
Otherwise, if T can not be made to conform to pt by instantiating its type variables, one determines all type variables in pt which are defined as type parameters of a method enclosing the pattern. Let the set of such type parameters be $b_1, \ldots, b_m$. Let $\mathcal{C}'_0$ be the subtype constraints reflecting the bounds of the type variables b_i. If T denotes an instance type of a final class, let $\mathcal{C}_2$ be the weakest set of subtype constraints over the type variables $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ such that $\mathcal{C}_0 \wedge \mathcal{C}'_0 \wedge \mathcal{C}_2$ implies that T conforms to pt. If T does not denote an instance type of a final class, let $\mathcal{C}_2$ be the weakest set of subtype constraints over the type variables $a_1, \ldots, a_n$ and $b_1, \ldots, b_m$ such that $\mathcal{C}_0 \wedge \mathcal{C}'_0 \wedge \mathcal{C}_2$ implies that it is possible to construct a type T' which conforms to both T and pt. It is a static error if there is no satisfiable set of constraints $\mathcal{C}_2$ with this property.

The final step consists in choosing type bounds for the type variables which imply the established constraint system. The process is different for the two cases above.

Case 1:
We take $a_i >: L_i <: U_i$ where each L_i is minimal and each U_i is maximal wrt $<:$ such that $a_i >: L_i <: U_i$ for $i = 1, \ldots, n$ implies $\mathcal{C}_0 \wedge \mathcal{C}_1$.

Case 2:
We take $a_i >: L_i <: U_i$ and $b_i >: L'_i <: U'_i$ where each L_i and L'_j is minimal and each U_i and U'_j is maximal such that $a_i >: L_i <: U_i$ for $i = 1, \ldots, n$ and $b_j >: L'_j <: U'_j$ for $j = 1, \ldots, m$ implies $\mathcal{C}_0 \wedge \mathcal{C}'_0 \wedge \mathcal{C}_2$.

In both cases, local type inference is permitted to limit the complexity of inferred bounds. Minimality and maximality of types have to be understood relative to the set of types of acceptable complexity.

Type parameter inference for constructor patterns.
Assume a constructor pattern $C(p_1, \ldots, p_n)$ where class C has type type parameters $a_1, \ldots, a_n$. These type parameters are inferred in the same way as for the typed pattern `(_: $C[a_1 , \ldots , a_n]$)`.

(79) *Consider the program fragment:*

```
val x: Any
x match {
  case y: List[a] => ...
}
```

Here, the type pattern `List[a]` *is matched against the expected type* `Any`*. The pattern binds the type variable* `a`*. Since* `List[a]` *conforms to* `Any` *for every type argument, there are no constraints on* `a`*. Hence,* `a` *is introduced as an abstract type with no bounds. The scope of* `a` *is right-hand side of its case clause.*

On the other hand, if `x` *is declared as*

```
val x: List[List[String]],
```

this generates the constraint `List[a] <: List[List[String]]`*, which simplifies to* `a <: List[String]`*, because* `List` *is covariant. Hence,* `a` *is introduced with upper bound* `List[String]`*.*

(80) *Consider the program fragment:*

```
val x: Any
x match {
  case y: List[String] => ...
}
```

Scala does not maintain information about type arguments at run-time, so there is no way to check that `x` *is a list of strings. Instead, the Scala compiler will erase the pattern to* `List[_]`*; that is, it will only test whether the top-level runtime-class of the value* `x` *conforms to* `List`*, and the pattern match will succeed if it does. This might lead to a class cast exception later on, in the case where the list* `x` *contains elements other than strings. The Scala compiler will flag this potential loss of type-safety with an "unchecked" warning message.*

(81) *Consider the program fragment*

```
class Term[A]
class Number(val n: Int) extends Term[Int]
def f[B](t: Term[B]): B = t match {
  case y: Number => y.n
}
```

The expected type of the pattern `y: Number` *is* `Term[B]`*. The type* `Number` *does not conform to* `Term[B]`*; hence Case 2 of the rules above applies. This means that* `b` *is treated as another type variable for which subtype constraints are inferred. In our case the applicable constraint is* `Number <: Term[B]`*, which entails* `B = Int`*. Hence,* `B` *is treated in the case clause as an abstract type with lower and upper bound* `Int`*. Therefore, the right hand side of the case clause,* `y.n`*, of type* `Int`*, is found to conform to the function's declared result type,* `Number`*.*

8.4 Pattern Matching Expressions

```
Expr            ::=  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
CaseClauses     ::=  CaseClause {CaseClause}
CaseClause      ::=  ‘case’ Pattern [Guard] ‘=>’ Block
```

A pattern matching expression

```
e match { case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$ }
```

consists of a selector expression e and a number $n > 0$ of cases. Each case consists of a (possibly guarded) pattern p_i and a block b_i. Each p_i might be complemented by a guard `if $e$` where e is a boolean expression. The scope of the pattern variables in p_i comprises the pattern's guard and the corresponding block b_i.

Let T be the type of the selector expression e and let $a_1, \ldots, a_m$ be the type parameters of all methods enclosing the pattern matching expression. For every a_i, let L_i be its lower bound and U_i be its higher bound. Every pattern $p \in \{p_1, , \ldots, p_n\}$ can be typed in two ways. First, it is attempted to type p with T as its expected type. If this fails, p is instead typed with a modified expected type T' which results from T by replacing every occurrence of a type parameter a_i by undefined. If this second step fails also, a compile-time error results. If the second step succeeds, let T_p be the type of pattern p seen as an expression. One then determines minimal bounds $L'_1, \ldots, L'_m$ and maximal bounds $U'_1, \ldots, U'_m$ such that for all i, $L_i <: L'_i$ and $U'_i <: U_i$ and the following constraint system is satisfied:

$$L_1 <: a_1 <: U_1 \;\wedge\; \ldots \;\wedge\; L_m <: a_m <: U_m \;\Rightarrow\; T_p <: T$$

If no such bounds can be found, a compile time error results. If such bounds are found, the pattern matching clause starting with p is then typed under the assumption that each a_i has lower bound L'_i instead of L_i and has upper bound U'_i instead of U_i.

The expected type of every block b_i is the expected type of the whole pattern matching expression. The type of the pattern matching expression is then the weak least upper bound of the types of all blocks b_i.

*When applying a pattern matching expression to a selector value, patterns are tried in sequence until one is found which matches the selector value. Say this case is **case**$p_i \Rightarrow b_i$. The result of the whole expression is then the result of evaluating b_i, where all pattern variables of p_i are bound to the corresponding parts of the selector value. If no matching pattern is found, a `scala.MatchError` exception is thrown.*

The pattern in a case may also be followed by a guard suffix `if e` with a boolean expression e. The guard expression is evaluated if the preceding pattern in the case matches. If the guard expression evaluates to `true`, the pattern match succeeds as normal. If the guard expression evaluates to `false`, the pattern in the case is considered not to match and the search for a matching pattern continues.

In the interest of efficiency the evaluation of a pattern matching expression may try patterns in some other order than textual sequence. This might affect evaluation through side effects in guards. However, it is guaranteed that a guard expression is evaluated only if the pattern it guards matches.

If the selector of a pattern match is an instance of a `sealed` class, the compilation of pattern matching can emit warnings which diagnose that a given set of patterns is not exhaustive, i.e. that there is a possibility of a `MatchError` being raised at run-time.

(82) *Consider the following definitions of arithmetic terms:*

```
abstract class Term[T]
case class Lit(x: Int) extends Term[Int]
case class Succ(t: Term[Int]) extends Term[Int]
case class IsZero(t: Term[Int]) extends Term[Boolean]
case class If[T](c: Term[Boolean],
                 t1: Term[T],
                 t2: Term[T]) extends Term[T]
```

There are terms to represent numeric literals, incrementation, a zero test, and a conditional. Every term carries as a type parameter the type of the expression it representes (either `Int` or `Boolean`).

A type-safe evaluator for such terms can be written as follows.

```
def eval[T](t: Term[T]): T = t match {
  case Lit(n)        => n
  case Succ(u)       => eval(u) + 1
  case IsZero(u)     => eval(u) == 0
  case If(c, u1, u2) => eval(if (eval(c)) u1 else u2)
}
```

Note that the evaluator makes crucial use of the fact that type parameters of enclosing methods can acquire new bounds through pattern matching.

For instance, the type of the pattern in the second case, `Succ(u)`, is `Int`. It conforms to the selector type T only if we assume an upper and lower bound of `Int` for T. Under the assumption `Int <: T <: Int` we can also verify that the type right hand side of the second case, `Int` conforms to its expected type, T.

8.5 Pattern Matching Anonymous Functions

```
BlockExpr ::= '{' CaseClauses '}'
```

An anonymous function can be defined by a sequence of cases

```
{ case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$ }
```

which appear as an expression without a prior `match`*. The expected type of such an expression must in part be defined. It must be either* `scala.Function$k$[$S_1 , \ldots , S_k$, $R$]` *for some* $k > 0$*, or* `scala.PartialFunction[$S_1$, $R$]`*, where the argument type(s)* $S_1, \ldots, S_k$ *must be fully determined, but the result type* R *may be undetermined.*

If the expected type is `scala.Function$k$[$S_1 , \ldots , S_k$, $R$]`*, the expression is taken to be equivalent to the anonymous function:*

```
($x_1: S_1 , \ldots , x_k: S_k$) => ($x_1 , \ldots , x_k$) match {
  case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$
}
```

Here, each x_i *is a fresh name. As was shown here, this anonymous function is in turn equivalent to the following instance creation expression, where* T *is the weak least upper bound of the types of all* b_i*.*

```
new scala.Function$k$[$S_1 , \ldots , S_k$, $T$] {
  def apply($x_1: S_1 , \ldots , x_k: S_k$): $T$ = ($x_1 , \ldots , x_k$) match {
    case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$
  }
}
```

If the expected type is `scala.PartialFunction[$S$, $R$]`*, the expression is taken to be equivalent to the following instance creation expression:*

```
new scala.PartialFunction[$S$, $T$] {
  def apply($x$: $S$): $T$ = x match {
    case $p_1$ => $b_1$ $\ldots$ case $p_n$ => $b_n$
  }
  def isDefinedAt($x$: $S$): Boolean = {
    case $p_1$ => true $\ldots$ case $p_n$ => true
    case _ => false
  }
}
```

Here, x *is a fresh name and* T *is the weak least upper bound of the types of all* b_i*. The final default case in the* `isDefinedAt` *method is omitted if one of the patterns* $p_1, \ldots, p_n$ *is already a variable or wildcard pattern.*

(83) Here is a method which uses a fold-left operation `/:` *to compute the scalar product of two vectors:*

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
    case (a, (b, c)) => a + b * c
  }
```

The case clauses in this code are equivalent to the following anonymous function:

```
  (x, y) => (x, y) match {
    case (a, (b, c)) => a + b * c
  }
```

Chapter 9

Top-Level Definitions

9.1 Compilation Units

```
CompilationUnit  ::=  {‘package’ QualId semi} TopStatSeq
TopStatSeq       ::=  TopStat {semi TopStat}
TopStat          ::=  {Annotation} {Modifier} TmplDef
                   |  Import
                   |  Packaging
                   |  PackageObject
                   |
QualId           ::=  id {‘.’ id}
```

A compilation unit consists of a sequence of packagings, import clauses, and class and object definitions, which may be preceded by a package clause.

A compilation unit

```
package $p_1$;
$\ldots$
package $p_n$;
$\mathit{stats}$
```

starting with one or more package clauses is equivalent to a compilation unit consisting of the packaging

```
package $p_1$ { $\ldots$
  package $p_n$ {
    $\mathit{stats}$
  } $\ldots$
}
```

Implicitly imported into every compilation unit are, in that order : the package `java.lang`*, the package* `scala`*, and the object* `scala.Predef`*. Members of a later import in that order hide members of an earlier import.*

9.2 Packagings

```
Packaging       ::=  'package' QualId [nl] '{' TopStatSeq '}'
```

A package is a special object which defines a set of member classes, objects and packages. Unlike other objects, packages are not introduced by a definition. Instead, the set of members of a package is determined by packagings.

A packaging `package $p$ { $\mathit{ds}$ }` *injects all definitions in ds as members into the package whose qualified name is p. Members of a package are called* top-level *definitions. If a definition in ds is labeled* **`private`**, *it is visible only for other members in the package.*

Inside the packaging, all members of package p are visible under their simple names. However this rule does not extend to members of enclosing packages of p that are designated by a prefix of the path p.

```
package org.net.prj {
  ...
}
```

all members of package `org.net.prj` *are visible under their simple names, but members of packages* `org` *or* `org.net` *require explicit qualification or imports.*

Selections $p.m$ from p as well as imports from p work as for objects. However, unlike other objects, packages may not be used as values. It is illegal to have a package with the same fully qualified name as a module or a class.

Top-level definitions outside a packaging are assumed to be injected into a special empty package. That package cannot be named and therefore cannot be imported. However, members of the empty package are visible to each other without qualification.

9.3 Package Objects

```
PackageObject   ::=  'package' 'object' ObjectDef
```

A package object `package object $p$ extends $t$` *adds the members of template t to the package p. There can be only one package object per package. The standard naming convention is to place the definition above in a file named* `package.scala` *that's located in the directory corresponding to package p.*

The package object should not define a member with the same name as one of the top-level objects or classes defined in package p. If there is a name conflict, the behavior of the program is currently undefined. It is expected that this restriction will be lifted in a future version of Scala.

9.4 Package References

```
QualId           ::=  id {'.' id}
```

A reference to a package takes the form of a qualified identifier. Like all other references, package references are relative. That is, a package reference starting in a name p will be looked up in the closest enclosing scope that defines a member named p.

The special predefined name `_root_` refers to the outermost root package which contains all top-level packages.

(84) *Consider the following program:*

```
package b {
  class B
}

package a.b {
  class A {
    val x = new _root_.b.B
  }
}
```

Here, the reference `_root_.b.B` refers to class `B` in the toplevel package `b`. If the `_root_` prefix had been omitted, the name `b` would instead resolve to the package `a.b`, and, provided that package does not also contain a class `B`, a compiler-time error would result.

9.5 Programs

A program *is a top-level object that has a member method* main *of type `(Array[String])Unit`. Programs can be executed from a command shell. The program's command arguments are are passed to the `main` method as a parameter of type `Array[String]`.*

The `main` method of a program can be directly defined in the object, or it can be inherited. The scala library defines a class `scala.Application` that defines an empty inherited `main` method. An objects m inheriting from this class is thus a program, which executes the initializaton code of the object m.

(85) *The following example will create a hello world program by defining a method* `main` *in module* `test.HelloWorld`*.*

```
package test
object HelloWord {
  def main(args: Array[String]) { println("hello world") }
}
```

This program can be started by the command ~~~~~~
scala test.HelloWorld ~~~~~~

In a Java environment, the command

```
java test.HelloWorld
```

would work as well.

`HelloWorld` *can also be defined without a* `main` *method by inheriting from* `Application` *instead:*

```
package test
object HelloWord extends Application {
  println("hello world")
}
```

Chapter 10

XML Expressions and Patterns

By Burak Emir
*This chapter describes the syntactic structure of XML expressions and patterns. It follows as closely as possible the XML 1.0 specification [**?**], changes being mandated by the possibility of embedding Scala code fragments.*

10.1 XML expressions

XML expressions are expressions generated by the following production, where the opening bracket '<' of the first element must be in a position to start the lexical XML mode.

```
XmlExpr ::= XmlContent {Element}
```

Well-formedness constraints of the XML specification apply, which means for instance that start tags and end tags must match, and attributes may only be defined once, with the exception of constraints related to entity resolution.

The following productions describe Scala's extensible markup language, designed as close as possible to the W3C extensible markup language standard. Only the productions for attribute values and character data are changed. Scala does not support declarations, CDATA sections or processing instructions. Entity references are not resolved at runtime.

~~~~~~

{.grammar} Element ::= EmptyElemTag | STag Content ETag

EmptyElemTag ::= '<' Name {S Attribute} [S] '/>'

STag ::= '<' Name {S Attribute} [S] '>'
ETag ::= ''{CharNoRef} ~~~~~~

XML expressions may contain Scala expressions as attribute values or within nodes. In the latter case, these are embedded using a single opening brace '{' and ended by a closing brace
~~~~~~

'}'. To express a single opening braces within XML text as generated by CharData, it must be doubled. Thus, '{{' represents the XML text '{' and does not introduce an embedded Scala expression.

```
BaseChar, Char, Comment, CombiningChar, Ideographic, NameChar, S, Reference
              ::=  $\mbox{\rm\em “as in W3C XML”}$

Char1         ::=  Char $\mbox{\rm\em without}$ ‘<’ | ‘&’
CharQ         ::=  Char1 $\mbox{\rm\em without}$ ‘"’
CharA         ::=  Char1 $\mbox{\rm\em without}$ ‘'’
CharB         ::=  Char1 $\mbox{\rm\em without}$ ‘{’

Name          ::=  XNameStart {NameChar}

XNameStart    ::= ‘_’ | BaseChar | Ideographic
                 $\mbox{\rm\em (as in W3C XML, but without }$ ‘:’
```

10.2 XML patterns

XML patterns are patterns generated by the following production, where the opening bracket '<' of the element patterns must be in a position to start the lexical XML mode.

```
XmlPattern  ::= ElementPattern
```

Well-formedness constraints of the XML specification apply.

An XML pattern has to be a single element pattern. It matches exactly those runtime representations of an XML tree that have the same structure as described by the pattern. XML patterns may contain Scala patterns.

Whitespace is treated the same way as in XML expressions. Patterns that are entity references, CDATA sections, processing instructions and comments match runtime representations which are the the same.

By default, beginning and trailing whitespace in element content is removed, and consecutive occurrences of whitespace are replaced by a single space character \u0020. This behavior can be changed to preserve all whitespace with a compiler option.

```
ElemPattern   ::=    EmptyElemTagP
                |    STagP ContentP ETagP

EmptyElemTagP ::=    ‘<’  Name [S] ‘/>’
STagP         ::=    ‘<’  Name [S] ‘>’
ETagP         ::=    ‘</’ Name [S] ‘>’
ContentP      ::=    [CharData] {(ElemPattern|ScalaPatterns) [CharData]}
```

```
ContentP1     ::=   ElemPattern
                |   Reference
                |   CDSect
                |   PI
                |   Comment
                |   ScalaPatterns
ScalaPatterns ::=   ‘{’ Patterns ‘}’
```

Chapter 11

User-Defined Annotations

```
Annotation       ::=  '@' SimpleType {ArgumentExprs}
ConstrAnnotation ::=  '@' SimpleType ArgumentExprs
```

User-defined annotations associate meta-information with definitions. A simple annotation has the form `@$c$` *or* `@$c(a_1 , \ldots , a_n)$`*. Here, c is a constructor of a class C, which must conform to the class* `scala.Annotation`*.*

Annotations may apply to definitions or declarations, types, or expressions. An annotation of a definition or declaration appears in front of that definition. An annotation of a type appears after that type. An annotation of an expression e appears after the expression e, separated by a colon. More than one annotation clause may apply to an entity. The order in which these annotations are given does not matter.

Examples:

```
@serializable class C { ... }          // A class annotation.
@transient @volatile var m: Int        // A variable annotation
String @local                          // A type annotation
(e: @unchecked) match { ... }          // An expression annotation
```

The meaning of annotation clauses is implementation-dependent. On the Java platform, the following annotations have a standard meaning.

- `@transient`
  Marks a field to be non-persistent; this is equivalent to the `transient` *modifier in Java.*
- `@volatile`
  Marks a field which can change its value outside the control of the program; this is equivalent to the `volatile` *modifier in Java.*

- `@serializable`
  Marks a class to be serializable; this is equivalent to inheriting from the `java.io.Serializable` *interface in Java.*

- `@SerialVersionUID(<longlit>)`
  Attaches a serial version identifier (a `long` *constant) to a class. This is equivalent to a the following field definition in Java:*

```
private final static SerialVersionUID = <longlit>
```

- `@throws(<classlit>)`
  A Java compiler checks that a program contains handlers for checked exceptions by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the `throws` *clause for the method or constructor must mention the class of that exception or one of the superclasses of the class of that exception.*

- `@deprecated(<stringlit>)`
  Marks a definition as deprecated. Accesses to the defined entity will then cause a deprecated warning mentioning the message `<stringlit>` *to be issued from the compiler. Deprecated warnings are suppressed in code that belongs itself to a definition that is labeled deprecated.*

- `@scala.reflect.BeanProperty`
  When prefixed to a definition of some variable X, this annotation causes getter and setter methods `getX`*,* `setX` *in the Java bean style to be added in the class containing the variable. The first letter of the variable appears capitalized after the* `get` *or* `set`*. When the annotation is added to the definition of an immutable value definition X, only a getter is generated. The construction of these methods is part of code-generation; therefore, these methods become visible only once a classfile for the containing class is generated.*

- `@scala.reflect.BooleanBeanProperty`
  This annotation is equivalent to `scala.reflect.BeanProperty`*, but the generated getter method is named* `isX` *instead of* `getX`*.*

- `@unchecked`
  When applied to the selector of a `match` *expression, this attribute suppresses any warnings about non-exhaustive pattern matches which would otherwise be emitted. For instance, no warnings would be produced for the method definition below.*

```
def f(x: Option[Int]) = (x: @unchecked) match {
  case Some(y) => y
}
```

  Without the `@unchecked` *annotation, a Scala compiler could infer that the pattern match is non-exhaustive, and could produce a warning because* `Option` *is a* `sealed` *class.*

- @uncheckedStable
  When applied a value declaration or definition, it allows the defined value to appear in a path, even if its type is volatile. For instance, the following member definitions are legal:

  ```
  type A { type T }
  type B
  @uncheckedStable val x: A with B // volatile type
  val y: x.T                        // OK since ‘x’ is still a path
  ```

  Without the @uncheckedStable annotation, the designator x would not be a path since its type A with B is volatile. Hence, the reference x.T would be malformed.

  When applied to value declarations or definitions that have non-volatile types, the annotation has no effect.

- @specialized
  When applied to the definition of a type parameter, this annotation causes the compiler to generate specialized definitions for primitive types. An optional list of primitive types may be given, in which case specialization takes into account only those types. For instance, the following code would generate specialized traits for Unit, Int and Double

  ```
  trait Function0[@specialized(Unit, Int, Double) T] {
    def apply: T
  }
  ```

  Whenever the static type of an expression matches a specialized variant of a definition, the compiler will instead use the specialized version. See [?] for more details of the implementation.

Other annotations may be interpreted by platform- or application-dependent tools. Class scala.Annotation has two sub-traits which are used to indicate how these annotations are retained. Instances of an annotation class inheriting from trait scala.ClassfileAnnotation will be stored in the generated class files. Instances of an annotation class inheriting from trait scala.StaticAnnotation will be visible to the Scala type-checker in every compilation unit where the annotated symbol is accessed. An annotation class can inherit from both scala.ClassfileAnnotation and scala.StaticAnnotation. If an annotation class inherits from neither scala.ClassfileAnnotation nor scala.StaticAnnotation, its instances are visible only locally during the compilation run that analyzes them.

Classes inheriting from scala.ClassfileAnnotation may be subject to further restrictions in order to assure that they can be mapped to the host environment. In particular, on both the Java and the .NET platforms, such classes must be toplevel; i.e. they may not be contained in another class or object. Additionally, on both Java and .NET, all constructor arguments must be constant expressions.

Chapter 12

The Scala Standard Library

The Scala standard library consists of the package `scala` *with a number of classes and modules. Some of these classes are described in the following.*

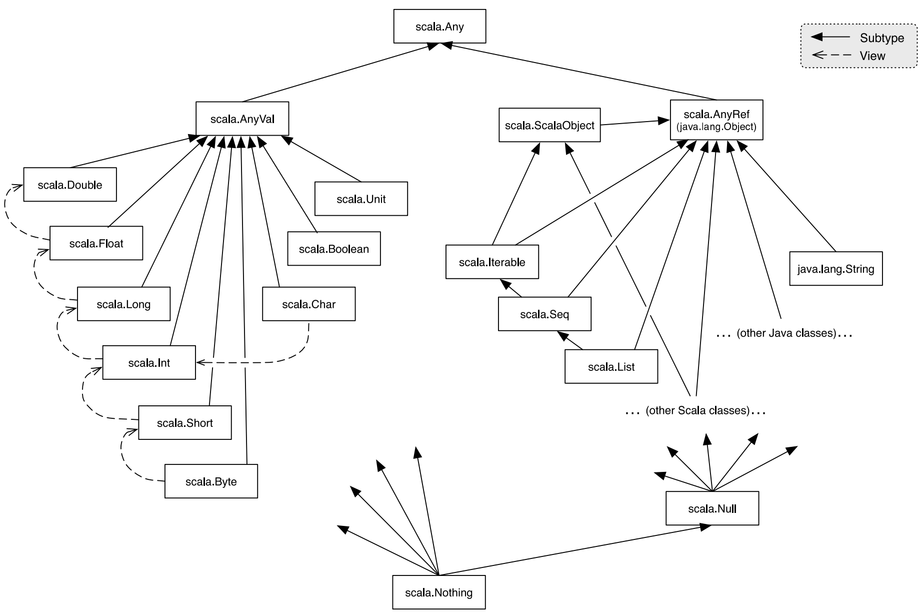

Figure 12.1: Class hierarchy of Scala

12.1 Root Classes

*Figure~**??** illustrates Scala's class hierarchy. The root of this hierarchy is formed by class* `Any`*. Every class in a Scala execution environment inherits directly or indirectly from this class. Class* `Any` *has two direct subclasses:* `AnyRef` *and AnyVal'.*

The subclass `AnyRef` *represents all values which are represented as objects in the underlying host system. Every user-defined Scala class inherits directly or indirectly from this class. Furthermore, every user-defined Scala class also inherits the trait* `scala.ScalaObject`*. Classes written in other languages still inherit from* `scala.AnyRef`*, but not from* `scala.ScalaObject`*.*

The class `AnyVal` *has a fixed number of subclasses, which describe values which are not implemented as objects in the underlying host system.*

Classes `AnyRef` *and* `AnyVal` *are required to provide only the members declared in class* `Any`*, but implementations may add host-specific methods to these classes (for instance, an implementation may identify class* `AnyRef` *with its own root class for objects).*

The signatures of these root classes are described by the following definitions.

```
package scala
/** The universal root class */
abstract class Any {

  /** Defined equality; abstract here */
  def equals(that: Any): Boolean

  /** Semantic equality between values */
  final def == (that: Any): Boolean  =
    if (null eq this) null eq that else this equals that

  /** Semantic inequality between values */
  final def != (that: Any): Boolean  =  !(this == that)

  /** Hash code; abstract here */
  def hashCode: Int = $\ldots$

  /** Textual representation; abstract here */
  def toString: String = $\ldots$

  /** Type test; needs to be inlined to work as given */
  def isInstanceOf[a]: Boolean

  /** Type cast; needs to be inlined to work as given */ */
  def asInstanceOf[A]: A = this match {
    case x: A => x
```

```
    case _ => if (this eq null) this
              else throw new ClassCastException()
  }
}

/** The root class of all value types */
final class AnyVal extends Any

/** The root class of all reference types */
class AnyRef extends Any {
  def equals(that: Any): Boolean      = this eq that
  final def eq(that: AnyRef): Boolean = $\ldots$ // reference equality
  final def ne(that: AnyRef): Boolean = !(this eq that)

  def hashCode: Int = $\ldots$     // hashCode computed from allocation address
  def toString: String  = $\ldots$ // toString computed from hashCode and class name

  def synchronized[T](body: => T): T // execute `body` in while locking `this`.
}

/** A mixin class for every user-defined Scala class */
trait ScalaObject extends AnyRef
```

The type test `$x$.isInstanceOf[$T$]` *is equivalent to a typed pattern match*

```
$x$ match {
  case _: $T'$ => true
  case _ => false
}
```

where the type T' is the same as T except if T is of the form D or $D[tps]$ where D is a type member of some outer class C. In this case T' is `$C$#$D$` *(or* `$C$#$D[tps]$`*, respectively), whereas T itself would expand to* `$C$.this.$D[tps]$`*. In other words, an* `isInstanceOf` *test does not check for the*

The test `$x$.asInstanceOf[$T$]` *is treated specially if T is a numeric value type. In this case the cast will be translated to an application of a conversion method* `x.to$T$`*. For non-numeric values x the operation will raise a* `ClassCastException`*.*

12.2 Value Classes

Value classes are classes whose instances are not represented as objects by the underlying host system. All value classes inherit from class `AnyVal`*. Scala implementations need to provide the value classes* `Unit`*,* `Boolean`*,* `Double`*,* `Float`*,* `Long`*,* `Int`*,* `Char`*,* `Short`*, and* `Byte` *(but are free to provide others as well). The signatures of these classes are defined in the following.*

12.2.1 Numeric Value Types

Classes `Double`*,* `Float`*,* `Long`*,* `Int`*,* `Char`*,* `Short`*, and* `Byte` *are together called* numeric value types. *Classes* `Byte`*,* `Short`*, or* `Char` *are called* subrange types. *Subrange types, as well as* `Int` *and* `Long` *are called* integer types, *whereas* `Float` *and* `Double` *are called* floating point types.

Numeric value types are ranked in the following partial order:

```
Byte - Short
             \
               Int - Long - Float - Double
             /
        Char
```

`Byte` *and* `Short` *are the lowest-ranked types in this order, whereas* `Double` *is the highest-ranked. Ranking does* not *imply a conformance relationship; for instance* `Int` *is not a subtype of* `Long`*. However, object* `Predef` *defines views from every numeric value type to all higher-ranked numeric value types. Therefore, lower-ranked types are implicitly converted to higher-ranked types when required by the context.*

Given two numeric value types S and T, the operation type *of S and T is defined as follows: If both S and T are subrange types then the operation type of S and T is* `Int`*. Otherwise the operation type of S and T is the larger of the two types wrt ranking. Given two numeric values v and w the operation type of v and w is the operation type of their run-time types.*

Any numeric value type T supports the following methods.

- *Comparison methods for equals (==), not-equals (!=), less-than (<), greater-than (>), less-than-or-equals (<=), greater-than-or-equals (>=), which each exist in 7 overloaded alternatives. Each alternative takes a parameter of some numeric value type. Its result type is type* `Boolean`*. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given comparison operation of that type.*
- *Arithmetic methods addition (+), subtraction (-), multiplication (*), division (/), and remainder (%), which each exist in 7 overloaded alternatives. Each alternative takes a parameter of some numeric value type U. Its result type is the operation type of T and U. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given arithmetic operation of that type.*
- *Parameterless arithmethic methods identity (+) and negation (-), with result type T. The first of these returns the receiver unchanged, whereas the second returns its negation.*
- *Conversion methods* `toByte`*,* `toShort`*,* `toChar`*,* `toInt`*,* `toLong`*,* `toFloat`*,* `toDouble` *which convert the receiver object to the target type, using the rules of Java's numeric type cast operation. The conversion might truncate the numeric value (as when going from* `Long` *to* `Int` *or from* `Int` *to* `Byte`*) or it might lose precision (as when going from* `Double` *to* `Float` *or when converting between* `Long` *and* `Float`*).*

Integer numeric value types support in addition the following operations:

- *Bit manipulation methods bitwise-and (&), bitwise-or {|}, and bitwise-exclusive-or (^), which each exist in 5 overloaded alternatives. Each alternative takes a parameter of some integer numeric value type. Its result type is the operation type of T and U. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given bitwise operation of that type.*
- *A parameterless bit-negation method (~). Its result type is the reciver type T or* `Int`*, whichever is larger. The operation is evaluated by converting the receiver to the result type and negating every bit in its value.*
- *Bit-shift methods left-shift (<<), arithmetic right-shift (>>), and unsigned right-shift (>>>). Each of these methods has two overloaded alternatives, which take a parameter n of type* `Int`*, respectively* `Long`*. The result type of the operation is the receiver type T, or* `Int`*, whichever is larger. The operation is evaluated by converting the receiver to the result type and performing the specified shift by n bits.*

Numeric value types also implement operations `equals`*,* `hashCode`*, and* `toString` *from class* `Any`*.*

The `equals` *method tests whether the argument is a numeric value type. If this is true, it will perform the == operation which is appropriate for that type. That is, the* `equals` *method of a numeric value type can be thought of being defined as follows:*

```
def equals(other: Any): Boolean = other match {
  case that: Byte   => this == that
  case that: Short  => this == that
  case that: Char   => this == that
  case that: Int    => this == that
  case that: Long   => this == that
  case that: Float  => this == that
  case that: Double => this == that
  case _ => false
}
```

The `hashCode` *method returns an integer hashcode that maps equal numeric values to equal results. It is guaranteed to be the identity for for type* `Int` *and for all subrange types.*

The `toString` *method displays its receiver as an integer or floating point number.*

(86) *As an example, here is the signature of the numeric value type* `Int`*:*

```
package scala
abstract sealed class Int extends AnyVal {
  def == (that: Double): Boolean  // double equality
```

```
  def == (that: Float): Boolean    // float equality
  def == (that: Long): Boolean     // long equality
  def == (that: Int): Boolean      // int equality
  def == (that: Short): Boolean    // int equality
  def == (that: Byte): Boolean     // int equality
  def == (that: Char): Boolean     // int equality
  /* analogous for !=, <, >, <=, >= */

  def + (that: Double): Double     // double addition
  def + (that: Float): Double      // float addition
  def + (that: Long): Long         // long addition
  def + (that: Int): Int           // int addition
  def + (that: Short): Int         // int addition
  def + (that: Byte): Int          // int addition
  def + (that: Char): Int          // int addition
  /* analogous for -, *, /, % */

  def & (that: Long): Long         // long bitwise and
  def & (that: Int): Int           // int bitwise and
  def & (that: Short): Int         // int bitwise and
  def & (that: Byte): Int          // int bitwise and
  def & (that: Char): Int          // int bitwise and
  /* analogous for |, ^ */

  def << (cnt: Int): Int           // int left shift
  def << (cnt: Long): Int          // long left shift
  /* analogous for >>, >>> */

  def unary_+ : Int                // int identity
  def unary_- : Int                // int negation
  def unary_~ : Int                // int bitwise negation

  def toByte: Byte                 // convert to Byte
  def toShort: Short               // convert to Short
  def toChar: Char                 // convert to Char
  def toInt: Int                   // convert to Int
  def toLong: Long                 // convert to Long
  def toFloat: Float               // convert to Float
  def toDouble: Double             // convert to Double
}
```

12.2.2 Class `Boolean`

Class `Boolean` has only two values: `true` and `false`. It implements operations as given in the following class definition.

```
package scala
abstract sealed class Boolean extends AnyVal {
  def && (p: => Boolean): Boolean = // boolean and
    if (this) p else false
  def || (p: => Boolean): Boolean = // boolean or
    if (this) true else p
  def &  (x: Boolean): Boolean =    // boolean strict and
    if (this) x else false
  def |  (x: Boolean): Boolean =    // boolean strict or
    if (this) true else x
  def == (x: Boolean): Boolean =    // boolean equality
    if (this) x else x.unary_!
  def != (x: Boolean): Boolean =    // boolean inequality
    if (this) x.unary_! else x
  def unary_!: Boolean =            // boolean negation
    if (this) false else true
}
```

The class also implements operations `equals`, `hashCode`, and `toString` from class `Any`.

The `equals` method returns `true` if the argument is the same boolean value as the receiver, `false` otherwise. The `hashCode` method returns a fixed, implementation-specific hash-code when invoked on `true`, and a different, fixed, implementation-specific hash-code when invoked on `false`. The `toString` method returns the receiver converted to a string, i.e. either `"true"` or `"false"`.

12.2.3 Class `Unit`

Class `Unit` has only one value: `()`. It implements only the three methods `equals`, `hashCode`, and `toString` from class `Any`.

The `equals` method returns `true` if the argument is the unit value `()`, `false` otherwise. The `hashCode` method returns a fixed, implementation-specific hash-code, The `toString` method returns `"()"`.

12.3 Standard Reference Classes

This section presents some standard Scala reference classes which are treated in a special way in Scala compiler – either Scala provides syntactic sugar for them, or the Scala compiler generates special code for their operations. Other classes in the standard Scala library are documented in the Scala library documentation by HTML pages.

12.3.1 Class String

Scala's `String` class is usually derived from the standard String class of the underlying host system (and may be identified with it). For Scala clients the class is taken to support in each case a method

```
def + (that: Any): String
```

which concatenates its left operand with the textual representation of its right operand.

12.3.2 The Tuple classes

Scala defines tuple classes `Tuple$n$` for $n = 2, \ldots, 9$. These are defined as follows.

```
package scala
case class Tuple$n$[+a_1, ..., +a_n](_1: a_1, ..., _$n$: a_$n$) {
  def toString = "(" ++ _1 ++ "," ++ $\ldots$ ++ "," ++ _$n$ ++ ")"
}
```

The implicitly imported `Predef` object defines the names `Pair` as an alias of `Tuple2` and `Triple` as an alias for `Tuple3`.

12.3.3 The Function Classes

Scala defines function classes `Function$n$` for $n = 1, \ldots, 9$. These are defined as follows.

```
package scala
trait Function$n$[-a_1, ..., -a_$n$, +b] {
  def apply(x_1: a_1, ..., x_$n$: a_$n$): b
  def toString = "<function>"
}
```

A subclass of `Function1` represents partial functions, which are undefined on some points in their domain. In addition to the `apply` method of functions, partial functions also have a `isDefined` method, which tells whether the function is defined at the given argument:

```
class PartialFunction[-A, +B] extends Function1[A, B] {
  def isDefinedAt(x: A): Boolean
}
```

The implicitly imported `Predef` object defines the name `Function` as an alias of `Function1`.

12.3.4 Class Array

The class of generic arrays is given as follows.

```
final class Array[A](len: Int) extends Seq[A] {
  def length: Int = len
  def apply(i: Int): A = $\ldots$
  def update(i: Int, x: A): Unit = $\ldots$
  def elements: Iterator[A] = $\ldots$
  def subArray(from: Int, end: Int): Array[A] = $\ldots$
  def filter(p: A => Boolean): Array[A] = $\ldots$
  def map[B](f: A => B): Array[B] = $\ldots$
  def flatMap[B](f: A => Array[B]): Array[B] = $\ldots$
}
```

If T is not a type parameter or abstract type, the type Array[T] is represented as the native array type `[]$T$` in the underlying host system. In that case `length` returns the length of the array, `apply` means subscripting, and `update` means element update. Because of the syntactic sugar for `apply` and `update` operations, we have the following correspondences between Scala and Java/C# code for operations on an array `xs`:

Scala	Java/C#
`xs.length`	`xs.length`
`xs(i)`	`xs[i]`
`xs(i) = e`	`xs[i] = e`

Arrays also implement the sequence trait `scala.Seq` by defining an `elements` method which returns all elements of the array in an `Iterator`.

Because of the tension between parametrized types in Scala and the ad-hoc implementation of arrays in the host-languages, some subtle points need to be taken into account when dealing with arrays. These are explained in the following.

First, unlike arrays in Java or C#, arrays in Scala are not *co-variant; That is, $S <: T$ does not imply `Array[$S$] $<:$ Array[$T$]` in Scala.*
However, it is possible to cast an array of S to an array of T if such a cast is permitted in the host environment.

For instance `Array[String]` does not conform to `Array[Object]`, even though `String` conforms to `Object`. However, it is possible to cast an expression of type `Array[String]` to `Array[Object]`, and this cast will succeed without raising a `ClassCastException`. Example:

```
val xs = new Array[String](2)
// val ys: Array[Object] = xs   // **** error: incompatible types
val ys: Array[Object] = xs.asInstanceOf[Array[Object]] // OK
```

Second, for polymorphic arrays*, that have a type parameter or abstract type T as their element type, a representation different from* `[]T` *might be used. However, it is guaranteed that* `isInstanceOf` *and* `asInstanceOf` *still work as if the array used the standard representation of monomorphic arrays:*

```
val ss = new Array[String](2)

def f[T](xs: Array[T]): Array[String] =
  if (xs.isInstanceOf[Array[String]]) xs.asInstanceOf[Array[String])
  else throw new Error("not an instance")

f(ss)                                     // returns ss
```

The representation chosen for polymorphic arrays also guarantees that polymorphic array creations work as expected. An example is the following implementation of method `mkArray`*, which creates an array of an arbitrary type T, given a sequence of T's which defines its elements.*

```
def mkArray[T](elems: Seq[T]): Array[T] = {
  val result = new Array[T](elems.length)
  var i = 0
  for (elem <- elems) {
    result(i) = elem
    i += 1
  }
}
```

Note that under Java's erasure model of arrays the method above would not work as expected – in fact it would always return an array of `Object`*.*

Third, in a Java environment there is a method `System.arraycopy` *which takes two objects as parameters together with start indices and a length argument, and copies elements from one object to the other, provided the objects are arrays of compatible element types.* `System.arraycopy` *will not work for Scala's polymorphic arrays because of their different representation. One should instead use method* `Array.copy` *which is defined in the companion object of class* `Array`*. This companion object also defines various constructor methods for arrays, as well as the extractor method* `unapplySeq` *which enables pattern matching over arrays.*

```
package scala
object Array {
```

```
  /** copies array elements from ‘src’ to ‘dest’. */
  def copy(src: AnyRef, srcPos: Int,
           dest: AnyRef, destPos: Int, length: Int): Unit = $\ldots$

  /** Concatenate all argument arrays into a single array. */
  def concat[T](xs: Array[T]*): Array[T] = $\ldots$

  /** Create a an array of successive integers. */
  def range(start: Int, end: Int): Array[Int] = $\ldots$

  /** Create an array with given elements. */
  def apply[A <: AnyRef](xs: A*): Array[A] = $\ldots$

  /** Analogous to above. */
  def apply(xs: Boolean*): Array[Boolean] = $\ldots$
  def apply(xs: Byte*)   : Array[Byte]    = $\ldots$
  def apply(xs: Short*)  : Array[Short]   = $\ldots$
  def apply(xs: Char*)   : Array[Char]    = $\ldots$
  def apply(xs: Int*)    : Array[Int]     = $\ldots$
  def apply(xs: Long*)   : Array[Long]    = $\ldots$
  def apply(xs: Float*)  : Array[Float]   = $\ldots$
  def apply(xs: Double*) : Array[Double]  = $\ldots$
  def apply(xs: Unit*)   : Array[Unit]    = $\ldots$

  /** Create an array containing several copies of an element. */
  def make[A](n: Int, elem: A): Array[A] = {

  /** Enables pattern matching over arrays */
  def unapplySeq[A](x: Array[A]): Option[Seq[A]] = Some(x)
}
```

(87) *The following method duplicates a given argument array and returns a pair consisting of the original and the duplicate:*

```
def duplicate[T](xs: Array[T]) = {
  val ys = new Array[T](xs.length)
  Array.copy(xs, 0, ys, 0, xs.length)
  (xs, ys)
}
```

12.4 Class Node

```
package scala.xml
```

```
trait Node {

  /** the label of this node */
  def label: String

  /** attribute axis */
  def attribute: Map[String, String]

  /** child axis (all children of this node) */
  def child: Seq[Node]

  /** descendant axis (all descendants of this node) */
  def descendant: Seq[Node] = child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  /** descendant axis (all descendants of this node) */
  def descendant_or_self: Seq[Node] = this::child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  override def equals(x: Any): Boolean = x match {
    case that:Node =>
      that.label == this.label &&
        that.attribute.sameElements(this.attribute) &&
          that.child.sameElements(this.child)
    case _ => false
  }

 /** XPath style projection function. Returns all children of this node
  *  that are labeled with 'that'. The document order is preserved.
  */
    def \(that: Symbol): NodeSeq = {
      new NodeSeq({
        that.name match {
          case "_" => child.toList
          case _ =>
            var res:List[Node] = Nil
            for (x <- child.elements if x.label == that.name) {
              res = x::res
            }
            res.reverse
        }
      })
    }
```

```
 /** XPath style projection function. Returns all nodes labeled with the
  *  name 'that' from the 'descendant_or_self' axis. Document order is preserved.
  */
  def \\(that: Symbol): NodeSeq = {
    new NodeSeq(
      that.name match {
        case "_" => this.descendant_or_self
        case _ => this.descendant_or_self.asInstanceOf[List[Node]].
        filter(x => x.label == that.name)
      })
  }

  /** hashcode for this XML node */
  override def hashCode =
    Utility.hashCode(label, attribute.toList.hashCode, child)

  /** string representation of this node */
  override def toString = Utility.toXML(this)

}
```

12.5 The Predef Object

The `Predef` object defines standard functions and type aliases for Scala programs. It is always implicitly imported, so that all its defined members are available without qualification. Its definition for the JVM environment conforms to the following signature:

```
package scala
object Predef {

  // classOf ---------------------------------------------------------

  /** Returns the runtime representation of a class type. */
  def classOf[T]: Class[T] = null
   // this is a dummy, classOf is handled by compiler.

  // Standard type aliases ---------------------------------------------

  type String    = java.lang.String
  type Class[T]  = java.lang.Class[T]

  // Miscellaneous -----------------------------------------------------

  type Function[-A, +B] = Function1[A, B]
```

```
type Map[A, +B] = collection.immutable.Map[A, B]
type Set[A] = collection.immutable.Set[A]

val Map = collection.immutable.Map
val Set = collection.immutable.Set

// Manifest types, companions, and incantations for summoning ---------

type ClassManifest[T] = scala.reflect.ClassManifest[T]
type Manifest[T]      = scala.reflect.Manifest[T]
type OptManifest[T]   = scala.reflect.OptManifest[T]
val ClassManifest     = scala.reflect.ClassManifest
val Manifest          = scala.reflect.Manifest
val NoManifest        = scala.reflect.NoManifest

def manifest[T](implicit m: Manifest[T])           = m
def classManifest[T](implicit m: ClassManifest[T]) = m
def optManifest[T](implicit m: OptManifest[T])     = m

// Minor variations on identity functions -----------------------------
def identity[A](x: A): A         = x    // @see ‘conforms‘ for the implicit version
def implicitly[T](implicit e: T) = e    // for summoning implicit values from the nether
@inline def locally[T](x: T): T  = x    // to communicate intent and avoid unmoored state

// Asserts, Preconditions, Postconditions -----------------------------

def assert(assertion: Boolean) {
  if (!assertion)
    throw new java.lang.AssertionError("assertion failed")
}

def assert(assertion: Boolean, message: => Any) {
  if (!assertion)
    throw new java.lang.AssertionError("assertion failed: " + message)
}

def assume(assumption: Boolean) {
  if (!assumption)
    throw new IllegalArgumentException("assumption failed")
}

def assume(assumption: Boolean, message: => Any) {
  if (!assumption)
    throw new IllegalArgumentException(message.toString)
}
```

```
def require(requirement: Boolean) {
  if (!requirement)
    throw new IllegalArgumentException("requirement failed")
}

def require(requirement: Boolean, message: => Any) {
  if (!requirement)
    throw new IllegalArgumentException("requirement failed: "+ message)
}


// tupling ---------------------------------------------------------

type Pair[+A, +B] = Tuple2[A, B]
object Pair {
  def apply[A, B](x: A, y: B) = Tuple2(x, y)
  def unapply[A, B](x: Tuple2[A, B]): Option[Tuple2[A, B]] = Some(x)
}

type Triple[+A, +B, +C] = Tuple3[A, B, C]
object Triple {
  def apply[A, B, C](x: A, y: B, z: C) = Tuple3(x, y, z)
  def unapply[A, B, C](x: Tuple3[A, B, C]): Option[Tuple3[A, B, C]] = Some(x)
}

// Printing and reading ---------------------------------------------

def print(x: Any) = Console.print(x)
def println() = Console.println()
def println(x: Any) = Console.println(x)
def printf(text: String, xs: Any*) = Console.printf(text.format(xs: _*))

def readLine(): String = Console.readLine()
def readLine(text: String, args: Any*) = Console.readLine(text, args)
def readBoolean() = Console.readBoolean()
def readByte() = Console.readByte()
def readShort() = Console.readShort()
def readChar() = Console.readChar()
def readInt() = Console.readInt()
def readLong() = Console.readLong()
def readFloat() = Console.readFloat()
def readDouble() = Console.readDouble()
def readf(format: String) = Console.readf(format)
def readf1(format: String) = Console.readf1(format)
def readf2(format: String) = Console.readf2(format)
```

```
  def readf3(format: String) = Console.readf3(format)

  // Implict conversions ------------------------------------------------

  ...
}
```

12.5.1 Predefined Implicit Definitions

The `Predef` object also contains a number of implicit definitions, which are available by default (because `Predef` is implicitly imported). Implicit definitions come in two priorities. High-priority implicits are defined in the `Predef` class itself whereas low priority implicits are defined in a class inherited by `Predef`. The rules of static overloading resolution stipulate that, all other things being equal, implicit resolution prefers high-priority implicits over low-priority ones.

The available low-priority implicits include definitions falling into the following categories.

1. *For every primitive type, a wrapper that takes values of that type to instances of a `runtime.Rich*` class. For instance, values of type `Int` can be implicitly converted to instances of class `runtime.RichInt`.*

2. *For every array type with elements of primitive type, a wrapper that takes the arrays of that type to instances of a `runtime.WrappedArray` class. For instance, values of type `Array[Float]` can be implicitly converted to instances of class `runtime.WrappedArray[Float]`. There are also generic array wrappers that take elements of type `Array[T]` for arbitrary `T` to `WrappedArray`s.*

3. *An implicit conversion from `String` to `WrappedString`.*

The available high-priority implicits include definitions falling into the following categories.

- *An implicit wrapper that adds `ensuring` methods with the following overloaded variants to type `Any`.*

```
def ensuring(cond: Boolean): A = { assert(cond); x }
def ensuring(cond: Boolean, msg: Any): A = { assert(cond, msg); x }
def ensuring(cond: A => Boolean): A = { assert(cond(x)); x }
def ensuring(cond: A => Boolean, msg: Any): A = { assert(cond(x), msg); x }
```

- *An implicit wrapper that adds a `->` method with the following implementation to type `Any`.*

```
def -> [B](y: B): (A, B) = (x, y)
```

- *For every array type with elements of primitive type, a wrapper that takes the arrays of that type to instances of a* `runtime.ArrayOps` *class. For instance, values of type* `Array[Float]` *can be implicitly converted to instances of class* `runtime.ArrayOps[Float]`*. There are also generic array wrappers that take elements of type* `Array[T]` *for arbitrary* `T` *to* `ArrayOps`*s.*

- *An implicit wrapper that adds* `+` *and* `formatted` *method with the following implementations to type* `Any`*.*

```
def +(other: String) = String.valueOf(self) + other
def formatted(fmtstr: String): String = fmtstr format self
```

- *Numeric primitive conversions that implement the transitive closure of the following mappings:*

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

- *Boxing and unboxing conversions between primitive types and their boxed versions:*

```
Byte    <-> java.lang.Byte
Short   <-> java.lang.Short
Char    <-> java.lang.Character
Int     <-> java.lang.Integer
Long    <-> java.lang.Long
Float   <-> java.lang.Float
Double  <-> java.lang.Double
Boolean <-> java.lang.Boolean
```

- *An implicit definition that generates instances of type* `T <:< T`*, for any type* `T`*. Here,* `<:<` *is a class defined as follows.*

```
sealed abstract class <:<[-From, +To] extends (From => To)
```

Implicit parameters of `<:<` *types are typically used to implement type constraints.*

Chapter 13

Scala Syntax Summary

The lexical syntax of Scala is given by the following grammar in EBNF form.

```
upper            ::=  ‘A’ | … | ‘Z’ | ‘\$’ | ‘_’  // and Unicode category Lu
lower            ::=  ‘a’ | … | ‘z’ // and Unicode category Ll
letter           ::=  upper | lower // and Unicode categories Lo, Lt, Nl
digit            ::=  ‘0’ | … | ‘9’
opchar           ::= // “all other characters in \u0020-\u007F and Unicode
                   // categories Sm, So except parentheses ([{}]) and periods”

op               ::=  opchar {opchar}
varid            ::=  lower idrest
plainid          ::=  upper idrest
                 |  varid
                 |  op
id               ::=  plainid
                 |  ‘\‘’ stringLit ‘\‘’
idrest           ::=  {letter | digit} [‘_’ op]

integerLiteral   ::=  (decimalNumeral | hexNumeral | octalNumeral) [‘L’ | ‘l’]
decimalNumeral   ::=  ‘0’ | nonZeroDigit {digit}
hexNumeral       ::=  ‘0’ ‘x’ hexDigit {hexDigit}
octalNumeral     ::=  ‘0’ octalDigit {octalDigit}
digit            ::=  ‘0’ | nonZeroDigit
nonZeroDigit     ::=  ‘1’ | … | ‘9’
octalDigit       ::=  ‘0’ | … | ‘7’

floatingPointLiteral
                 ::=  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’

booleanLiteral   ::=  ‘true’ | ‘false’

characterLiteral ::=  ‘\’‘ printableChar ‘\’’
                 |  ‘\’ charEscapeSeq ‘\’’

stringLiteral    ::=  ‘”’ {stringElement} ‘”’
                 |  ‘”””’ multiLineChars ‘”””’
stringElement    ::=  printableCharNoDoubleQuote
                 |  charEscapeSeq
multiLineChars   ::=  {[‘”’] [‘”’] charNoDoubleQuote} {‘”’}

symbolLiteral    ::=  ‘’’ plainid

comment          ::=  ‘/*’ “any sequence of characters” ‘*/’
                 |  ‘//’ “any sequence of characters up to end of line”

nl               ::=  $\mathit{“new line character”}$
semi             ::=  ‘;’ |  nl {nl}
```

The context-free syntax of Scala is given by the following EBNF grammar.

```
Literal           ::=  [‘-’] integerLiteral
                    |  [‘-’] floatingPointLiteral
                    |  booleanLiteral
                    |  characterLiteral
                    |  stringLiteral
                    |  symbolLiteral
                    |  ‘null’

QualId            ::=  id {‘.’ id}
ids               ::=  id {‘,’ id}

Path              ::=  StableId
                    |  [id ‘.’] ‘this’
StableId          ::=  id
                    |  Path ‘.’ id
                    |  [id ‘.’] ‘super’ [ClassQualifier] ‘.’ id
ClassQualifier    ::=  ‘[’ id ‘]’

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’
                   |  ‘(’ Types ‘)’
TypeArgs          ::=  ‘[’ Types ‘]’
Types             ::=  Type {‘,’ Type}
Refinement        ::=  [nl] ‘{’ RefineStat {semi RefineStat} ‘}’
RefineStat        ::=  Dcl
                   |  ‘type’ TypeDef
                   |
TypePat           ::=  Type

Ascription        ::=  ‘:’ InfixType
                   |  ‘:’ Annotation {Annotation}
                   |  ‘:’ ‘_’ ‘*’

Expr              ::=  (Bindings | [‘implicit’] id | ‘_’) ‘=>’ Expr
                   |  Expr1
Expr1             ::=  ‘if’ ‘(’ Expr ‘)’ {nl} Expr [[semi] else Expr]
                   |  ‘while’ ‘(’ Expr ‘)’ {nl} Expr
                  |  ‘try’ ‘{’ Block ‘}’ [‘catch’  ‘{’ CaseClauses ‘}’]
                      [‘finally’ Expr]
                   |  ‘do’ Expr [semi] ‘while’ ‘(’ Expr ‘)’
                   |  ‘for’ (‘(’ Enumerators ‘)’ | ‘{’ Enumerators ‘}’)
                      {nl} [‘yield’] Expr
                   |  ‘throw’ Expr
                   |  ‘return’ [Expr]
                   |  [SimpleExpr ‘.’] id ‘=’ Expr
                   |  SimpleExpr1 ArgumentExprs ‘=’ Expr
                   |  PostfixExpr
                   |  PostfixExpr Ascription
                   |  PostfixExpr ‘match’ ‘{’ CaseClauses ‘}’
PostfixExpr       ::=  InfixExpr [id [nl]]
InfixExpr         ::=  PrefixExpr
                   |  InfixExpr id [nl] InfixExpr
```

```
PrefixExpr        ::=  [‘-’ | ‘+’ | ‘~’ | ‘!’] SimpleExpr
SimpleExpr        ::=  ‘new’ (ClassTemplate | TemplateBody)
                    |  BlockExpr
                    |  SimpleExpr1 [‘_’]
SimpleExpr1       ::=  Literal
                    |  Path
                    |  ‘_’
                    |  ‘(’ [Exprs] ‘)’
                    |  SimpleExpr ‘.’ id
                    |  SimpleExpr TypeArgs
                    |  SimpleExpr1 ArgumentExprs
                    |  XmlExpr
Exprs             ::=  Expr {‘,’ Expr}
ArgumentExprs     ::=  ‘(’ [Exprs] ‘)’
                    |  ‘(’ [Exprs ‘,’] PostfixExpr ‘:’ ‘_’ ‘*’ ‘)’
                    |  [nl] BlockExpr
BlockExpr         ::=  ‘{’ CaseClauses ‘}’
                    |  ‘{’ Block ‘}’
Block             ::=  {BlockStat semi} [ResultExpr]
BlockStat         ::=  Import
                    |  {Annotation} [‘implicit’ | ‘lazy’] Def
                    |  {Annotation} {LocalModifier} TmplDef
                    |  Expr1
                    |
ResultExpr        ::=  Expr1
             |  (Bindings | ([‘implicit’] id | ‘_’) ‘:’ CompoundType) ‘=>’ Block

Enumerators       ::=  Generator {semi Enumerator}
Enumerator        ::=  Generator
                    |  Guard
                    |  ‘val’ Pattern1 ‘=’ Expr
Generator         ::=  Pattern1 ‘<-’ Expr [Guard]

CaseClauses       ::=  CaseClause { CaseClause }
CaseClause        ::=  ‘case’ Pattern [Guard] ‘=>’ Block
Guard             ::=  ‘if’ PostfixExpr

Pattern           ::=  Pattern1 { ‘|’ Pattern1 }
Pattern1          ::=  varid ‘:’ TypePat
                    |  ‘_’ ‘:’ TypePat
                    |  Pattern2
Pattern2          ::=  varid [‘@’ Pattern3]
                    |  Pattern3
Pattern3          ::=  SimplePattern
                    |  SimplePattern { id [nl] SimplePattern }
SimplePattern     ::=  ‘_’
```

```
                     |  varid
                     |  Literal
                     |  StableId
                     |  StableId ‘(’ [Patterns ‘)’
                     |  StableId ‘(’ [Patterns ‘,’] [varid ‘@’] ‘_’ ‘*’ ‘)’
                     |  ‘(’ [Patterns] ‘)’
                     |  XmlPattern
Patterns          ::=  Pattern [‘,’ Patterns]
                     |  ‘_’ *

TypeParamClause   ::=  ‘[’ VariantTypeParam {‘,’ VariantTypeParam} ‘]’
FunTypeParamClause::=  ‘[’ TypeParam {‘,’ TypeParam} ‘]’
VariantTypeParam  ::=  {Annotation} [‘+’ | ‘-’] TypeParam
TypeParam         ::=  (id | ‘_’) [TypeParamClause] [‘>:’ Type] [‘<:’ Type]
                       {‘<%’ Type} {‘:’ Type}
ParamClauses      ::=  {ParamClause} [[nl] ‘(’ ‘implicit’ Params ‘)’]
ParamClause       ::=  [nl] ‘(’ [Params] ‘)’
Params            ::=  Param {‘,’ Param}
Param             ::=  {Annotation} id [‘:’ ParamType] [‘=’ Expr]
ParamType         ::=  Type
                     |  ‘=>’ Type
                     |  Type ‘*’
ClassParamClauses ::=  {ClassParamClause}
                       [[nl] ‘(’ ‘implicit’ ClassParams ‘)’]
ClassParamClause  ::=  [nl] ‘(’ [ClassParams] ‘)’
ClassParams       ::=  ClassParam {‘’ ClassParam}
ClassParam        ::=  {Annotation} [{Modifier} (‘val’ | ‘var’)]
                       id ‘:’ ParamType [‘=’ Expr]
Bindings          ::=  ‘(’ Binding {‘,’ Binding ‘)’
Binding           ::=  (id | ‘_’) [‘:’ Type]

Modifier          ::=  LocalModifier
                     |  AccessModifier
                     |  ‘override’
LocalModifier     ::=  ‘abstract’
                     |  ‘final’
                     |  ‘sealed’
                     |  ‘implicit’
                     |  ‘lazy’
AccessModifier    ::=  (‘private’ | ‘protected’) [AccessQualifier]
AccessQualifier   ::=  ‘[’ (id | ‘this’) ‘]’

Annotation        ::=  ‘@’ SimpleType {ArgumentExprs}
ConstrAnnotation  ::=  ‘@’ SimpleType ArgumentExprs
NameValuePair     ::=  ‘val’ id ‘=’ PrefixExpr
```

```
TemplateBody     ::= [nl] ‘{’ [SelfType] TemplateStat {semi TemplateStat} ‘}’
TemplateStat     ::= Import
                   | {Annotation [nl]} {Modifier} Def
                   | {Annotation [nl]} {Modifier} Dcl
                   | Expr
                   |
SelfType         ::= id [‘:’ Type] ‘=>’
                   | ‘this’ ‘:’ Type ‘=>’

Import           ::= ‘import’ ImportExpr {‘,’ ImportExpr}
ImportExpr       ::= StableId ‘.’ (id | ‘_’ | ImportSelectors)
ImportSelectors  ::= ‘{’ {ImportSelector ‘,’} (ImportSelector | ‘_’) ‘}’
ImportSelector   ::= id [‘=>’ id | ‘=>’ ‘_’]

Dcl              ::= ‘val’ ValDcl
                   | ‘var’ VarDcl
                   | ‘def’ FunDcl
                   | ‘type’ {nl} TypeDcl

ValDcl           ::= ids ‘:’ Type
VarDcl           ::= ids ‘:’ Type
FunDcl           ::= FunSig [‘:’ Type]
FunSig           ::= id [FunTypeParamClause] ParamClauses
TypeDcl          ::= id [TypeParamClause] [‘>:’ Type] [‘<:’ Type]

PatVarDef        ::= ‘val’ PatDef
                   | ‘var’ VarDef
Def              ::= PatVarDef
                   | ‘def’ FunDef
                   | ‘type’ {nl} TypeDef
                   | TmplDef
PatDef           ::= Pattern2 {‘,’ Pattern2} [‘:’ Type] ‘=’ Expr
VarDef           ::= PatDef
                   | ids ‘:’ Type ‘=’ ‘_’
FunDef           ::= FunSig [‘:’ Type] ‘=’ Expr
                   | FunSig [nl] ‘{’ Block ‘}’
                   | ‘this’ ParamClause ParamClauses
                     (‘=’ ConstrExpr | [nl] ConstrBlock)
TypeDef          ::= id [TypeParamClause] ‘=’ Type

TmplDef          ::= [‘case’] ‘class’ ClassDef
                   | [‘case’] ‘object’ ObjectDef
                   | ‘trait’ TraitDef
ClassDef         ::= id [TypeParamClause] {ConstrAnnotation} [AccessModifier]
                     ClassParamClauses ClassTemplateOpt
TraitDef         ::= id [TypeParamClause] TraitTemplateOpt
```

```
ObjectDef         ::=  id ClassTemplateOpt
ClassTemplateOpt  ::=  ‘extends’ ClassTemplate | [[‘extends’] TemplateBody]
TraitTemplateOpt  ::=  ‘extends’ TraitTemplate | [[‘extends’] TemplateBody]
ClassTemplate     ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate     ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents      ::=  Constr {‘with’ AnnotType}
TraitParents      ::=  AnnotType {‘with’ AnnotType}
Constr            ::=  AnnotType {ArgumentExprs}
EarlyDefs         ::= ‘{’ [EarlyDef {semi EarlyDef}] ‘}’ ‘with’
EarlyDef          ::=  {Annotation [nl]} {Modifier} PatVarDef

ConstrExpr        ::=  SelfInvocation
                    |  ConstrBlock
ConstrBlock       ::=  ‘{’ SelfInvocation {semi BlockStat} ‘}’
SelfInvocation    ::=  ‘this’ ArgumentExprs {ArgumentExprs}

TopStatSeq        ::=  TopStat {semi TopStat}
TopStat           ::=  {Annotation [nl]} {Modifier} TmplDef
                    |  Import
                    |  Packaging
                    |  PackageObject
                    |
Packaging         ::=  ‘package’ QualId [nl] ‘{’ TopStatSeq ‘}’
PackageObject     ::=  ‘package’ ‘object’ ObjectDef

CompilationUnit   ::=  {‘package’ QualId semi} TopStatSeq
```

Chapter 14

References

Kennedy, Andrew J., and Benjamin C. Pierce. 2007. "On Decidability of Nominal Subtyping with Variance."

Odersky, Martin. 2006. "The Scala Experiment – Can We Provide Better Language Support for Component Systems?" In Proc. ACM Symposium on Principles of Programming Languages.

Odersky, Martin, Vincent Cremet, Christine Röckl, and Matthias Zenger. 2003. "A Nominal Theory of Objects with Dependent Types." In Proc. ECOOP'03.

Odersky, Martin, and Matthias Zenger. 2005a. "Scalable Component Abstractions." In Proc. OOPSLA.

———. 2005b. "Independently Extensible Solutions to the Expression Problem." In Proc. FOOL 12.

Odersky, Martin, and al. 2004. An Overview of the Scala Programming Language.