The Scala Language Specification
Version 1.0

DRAFT
July 27, 2004

Martin Odersky
Philippe Altherr
Vincent Cremet
Burak Emir
Stéphane Micheloud
Nikolay Mihaylov
Michel Schinz
Erik Stenman
Matthias Zenger

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

Contents

I Rationale

There are hundreds of programming languages in active use, and many more are being designed each year. It is therefore hard to justify the development of yet another language. Nevertheless, this is what we attempt to do here. Our argument is based on two claims:

> *Claim 1:* The raise in importance of web services and other distributed software is a fundamental paradigm shift in programming. It is comparable in scale to the shift 20 years ago from character-oriented to graphical user interfaces.
>
> *Claim 2:* That paradigm shift will provide demand for new programming languages, just as graphical user interfaces promoted the adoption of object-oriented languages.

For the last 20 years, the most common programming model was object-oriented: System components are objects, and computation is done by method calls. Methods themselves take object references as parameters. Remote method calls let one extend this programming model to distributed systems. The problem of this model is that it does not scale up very well to wide-scale networks where messages can be delayed and components may fail. Web services address the message delay problem by increasing granularity, using method calls with larger, structured arguments, such as XML trees. They address the failure problem by using transparent replication and avoiding server state. Conceptually, they are *tree transformers* that consume incoming message documents and produce outgoing ones.

Why should this have an effect on programming languages? There are at least two reasons: First, today's object-oriented languages are not very good at analyzing and transforming XML trees. Because such trees usually contain data but no methods, they have to be decomposed and constructed from the "outside", that is from code which is external to the tree definition itself. In an object-oriented language, the ways of doing so are limited. The most common solution [W3Ca] is to represent trees in a generic way, where all tree nodes are values of a common type. This makes it easy to write generic traversal functions, but forces applications to operate on a very low conceptual level, which often loses important semantic distinctions present in the XML data. More semantic precision is obtained if different internal types model different kinds of nodes. But then tree decompositions require the use of run-time type tests and type casts to adapt the treatment to the kind of node encountered. Such type tests and type casts are generally not considered good object-oriented style. They are rarely efficient, nor easy to use.

Conceivably, the glue problem could be addressed by a "multi-paradigm" language that would express object-oriented, concurrent, as well as functional aspects of an application. But one needs to be careful not to simply replace cross-language glue by awkward interfaces between different paradigms within the language itself. Ideally, one would hope for a fusion which unifies concepts found in different paradigms instead of an agglutination, which merely includes them side by side.

This fusion is what we try to achieve with Scala [1].

Scala is both an object-oriented and functional language. 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 work seamlessly with mainstream object-oriented languages, in particular Java and C#.

Scala is also 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. Furthermore, this notion of pattern matching naturally extends to the processing of XML data.

The design of Scala is driven by the desire to unify object-oriented and functional elements. Here are three examples how this is achieved:

- Since every function is a value and every value is an object, it follows that every function in Scala is an object. Indeed, there is a root class for functions which is specialized in the Scala standard library to data structures such as arrays and hash tables.
- Data structures in many functional languages are defined using algebraic data types. They are decomposed using pattern matching. Object-oriented languages, on the other hand, describe data with class hierarchies. Algebraic data types are usually closed, in that the range of alternatives of a type is fixed when the type is defined. By contrast, class hierarchies can be extended by adding new leaf classes. Scala adopts the object-oriented class hierarchy scheme for data definitions, but allows pattern matching against values coming from a whole class hierarchy, not just values of a single type. This can express both closed and extensible data types, and also provides a convenient way to exploit run-time type information in cases where static typing is too restrictive.
- Module systems of functional languages such as SML or Caml excel in abstraction; they allow very precise control over visibility of names and types, including the ability to partially abstract over types. By contrast, object-oriented languages excel in composition; they offer several composition mechanisms lacking in module systems, including inheritance and unlimited recursion between objects and classes. Scala unifies the notions of object and module, of module signature and interface, as well as of functor and class. This combines the abstraction facilities of functional module systems with the composition constructs of object-oriented languages. The unification is made possible by means of a new type system based on path-dependent types [OCRZ03].

There are several other languages that try to bridge the gap between the functional and object oriented paradigms. Smalltalk[GR83], Python[vRD03], or Ruby[Mat01]

[1] Scala stands for “Scalable Language”. The term means “Stairway” in Italian

come to mind. Unlike these languages, Scala has an advanced static type system, which contains several innovative constructs. This aspect makes the Scala definition a bit more complicated than those of the languages above. On the other hand, Scala enjoys the robustness, safety and scalability benefits of strong static typing. Furthermore, Scala incorporates recent advances in type inference, so that excessive type annotations in user programs can usually be avoided.

II The Scala Language Specification

Chapter 1

Lexical Syntax

Scala programs are written using the Unicode character set. 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), title-case 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

Syntax:

```
op        ::=  special {special}
varid     ::=  lower idrest
id        ::=  upper idrest
             |  varid
             |  op
             |  ‘‘‘string chars‘’’
idrest    ::= {letter | digit} {’_’ (op | idrest)}
```

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 other string composed of either letters and digits or of special characeters. Second, an identifier can start with a special character followed by an arbitrary sequence of special characters. 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). As usual, a longest match rule applies. For instance, the string

```
big_bob++=z3
```

decomposes into the three identifiers `big_bob`, `++=`, and `z3`. The rules for pattern matching further distinguish between *variable identifiers*, which start with a lower case letter, and *constant identifiers*, which do not.

The ‘$’ character is reserved for compiler-synthesized identifiers. User programs are not allowed to define identifiers which contain ‘$’ 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    if    import    new
null    object    override    package    private
protected    return    sealed    super    this
throw    trait    try    true    type
val    var    while    with    yield
_    :    =    =>    <-    <:    >:    #    @
```

The Unicode operator \u21D2 ‘⇒’ has the ASCII equivalent ‘=>’, which is also reserved.

Example 1.1.1 Here are examples of identifiers:

```
x    Object        maxIndex        p2p      empty_?
```

```
+    +_field       αρετη
```

1.2 Braces and Semicolons

A semicolon ';' is implicitly inserted after every closing brace if there is a new line character between closing brace and the next regular token after it, except if that token cannot legally start a statement.

The tokens which cannot legally start a statement are the following delimiters and reserved words:

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

1.3 Literals

There are literals for integer numbers (of types `Int` and `Long`), floating point numbers (of types `Float` and `Double`), characters, and strings. The syntax of these literals is in each case as in Java.

Syntax:

```
intLit       ::=  "as in Java"
floatLit     ::=  "as in Java"
charLit      ::=  "as in Java"
stringLit    ::=  "as in Java"
```

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.

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.

Syntax:

```
( 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 successully 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.

Chapter 2

Identifiers, Names and Scopes

Names in Scala identify types, values, methods, and classes which are collectively called *entities*. Names are introduced by definitions, declarations (§4) or import clauses (§4.7), which are collectively called *binders*.

There are two different name spaces, one for types (§3) and one for terms (§6). The same name may designate a type and a term, depending on the context where the name is used.

A definition or declaration has a *scope* in which the entity defined by a single name can be accessed using a simple name. Scopes are nested, and a definition or declaration in some inner scope *shadows* a definition in an outer scope that contributes to the same name space. Furthermore, a definition or declaration shadows bindings introduced by a preceding import clause, even if the import clause is in the same block. Import clauses, on the other hand, only shadow bindings introduced by other import clauses in outer blocks.

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

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

It is an error if no such binder 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 binder. In that case, the type of x is the type of the referenced entity.

Example 2.0.1 Consider the following nested definitions and imports:

```
object m1 {
```

```
  object m2 { val x: int = 1; val y: int = 2 }
  object m3 { val x: boolean = true; val y: String = "" }
  val x: int = 3;
  { import m2._;            // shadows nothing
                            //   reference to 'x' is ambiguous here
    val x: String = "abc"; // shadows preceding import
                            //   name 'x' refers to latest val definition
    { import m3._           // shadows only preceding import m2
                            // reference to 'x' is ambiguous here
                            //   name 'y' refers to latest import clause
    }
  }
}
```

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 (§3.2). The type of $e.x$ is the member type of the referenced entity in T.

Chapter 3

Types

Syntax:

```
Type          ::=  Type1 '=>' Type
                |  '(' [Types] ')' '=>' Type
                |  Type1
Type1         ::=  SimpleType {with SimpleType} [Refinement]
SimpleType    ::=  StableId
                |  SimpleType '#' id
                |  Path '.' type
                |  SimpleType TypeArgs
                |  '(' Type ')'
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 (§3.2.3) that refers to a class[1] (§5.2), or as a *compound type* (§3.2.5) consisting of class types and possibly also a refinement (§3.2.5) that further constrains the types of its members.

A shorthand exists for denoting function types (§3.2.6). Abstract value types are introduced by type parameters and abstract type bindings (§4.3). Parentheses in types are used for grouping.

Non-value types capture properties of identifiers that are not values (§3.3). There is no syntax to express these types directly in Scala.

[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).

3.1 Paths

Syntax:

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

Paths are not types themselves, but they can be a part of named types and in that way 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 members introduced by value or object definitions, as well as packages.
- 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 mixin 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

3.2.1 Singleton Types

Syntax:

```
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 the value denoted by p and `null`.

3.2.2 Type Projection

Syntax:

```
SimpleType  ::=  SimpleType '#' id
```

A type projection $T\#x$ references the type member named x of type T. T must be either a singleton type, or a non-abstract class type, or a Java class type (in either of the last two cases, it is guaranteed that T has no abstract type members).

3.2.3 Type Designators

Syntax:

```
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 ϵ`.type#`t.

A qualified type designator has the form p`.`t where p is a path (§3.1) and t is a type name. Such a type designator is equivalent to the type projection p`.type#`x.

Example 3.2.1 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                    ϵ.type#t
Int                  scala.type#Int
scala.Int            scala.type#Int
data.maintable.Node  data.maintable.type#Node
```

3.2.4 Parameterized Types

Syntax:

```
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 $a_1, \ldots, a_n$ with 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. $L_i\sigma <: T_i <: U_i\sigma$ where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$.

Example 3.2.2 Given the partial type definitions:

```
class TreeMap[a <: Ord[a], b] { ... }
class List[a] { ... }
class I extends Ord[I] { ... }
```

the following parameterized types are well formed:

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

Example 3.2.3 Given the type definitions of Example 3.2.2, the following types are ill-formed:

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

3.2.5 Compound Types

Syntax:

```
Type            ::=  SimpleType {with SimpleType} [Refinement]
Refinement      ::=  '{' [RefineStat {';' 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 $\{R\}$. Each component type T_i must be a class type . A refinement $\{R\}$ contains declarations and type definitions. Each declaration or definition in a refinement must override a declaration or definition in one of the component types $T_1, \ldots, T_n$. The usual rules for overriding (§5.1.5) apply. 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 `{}`.

3.2.6 Function Types

Syntax:

```
SimpleType      ::=  Type1 '=>' Type
                  |  '(' [Types] ')' '=>' Type
```

The type `(`$T_1, \ldots, T_n$`) => `U represents the set of function values that take arguments of types $T_1, \ldots, T_n$ and yield results of type U. In the case of exactly one argument type T `=>` U is a shorthand for `(`T`) => `U. Function types associate to the right, e.g. `(`S`) => (`T`) => `U is the same as `(`S`) => ((`T`) => `U`)`.

Function types are shorthands for class types that define `apply` functions. Specifically, the n-ary function type `(`$T_1, \ldots, T_n$`) => U` is a shorthand for the class type `Function`n`[`$T_1, \ldots, T_n, U$`]`. Such class types are defined in the Scala library for n between 0 and 9 as follows.

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

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

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 $(Ts)U$, where (Ts) is a sequence of types $(T_1, \ldots, T_n)$ for some $n \geq 0$ and U is a (value or method) type. This type represents named methods that take arguments of types $T_1, \ldots, T_n$ and that return a result of type U.

Method types associate to the right: $(Ts_1)(Ts_2)U$ is treated as $(Ts_1)((Ts_2)U)$.

A special case are types of methods without any parameters. They are written here $[]T$, following the syntax for polymorphic method types (§3.3.2). 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 (§3.7).

Example 3.3.1 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 `[`*tps*`]`T where `[`*tps*`]` is a type parameter section `[`a_1 `>:` L_1 `<:` $U_1, \ldots, a_n$ `>:` L_n `<:` U_n`]` for some $n \geq 0$ and T is a (value or method) type. This type represents named methods that take type arguments $S_1, \ldots, S_n$ which conform (§3.2.4) to the lower bounds $L_1, \ldots, L_n$ and the upper bounds $U_1, \ldots, U_n$ and that yield results of type T.

Example 3.3.2 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 >: All <: Any] List[a]
union : [a >: All <: Comparable[a]] (x: Set[a], xs: Set[a]) Set[a]  .
```

3.4 Base Classes and Member Definitions

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

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

These notions are defined mutually recursively as follows.

1. The set of *base classes* of a type is a set of class types, given as follows.

- The base classes of a class type C are the base classes of class C.
- The base classes of an aliased type are the base classes of its alias.
- The base classes of an abstract type are the base classes of its upper bound.
- The base classes of a parameterized type C`[`$T_1, \ldots, T_n$`]` are the base classes 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 classes of a singleton type p`.type` are the base classes of the type of p.
- The base classes 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 classes of all T_i's. If $\mathscr{S}$ contains several

type instances of the same class, say $S^i\#C[T_1^i, \ldots, T_n^i]$ $(i \in I)$, then all those instances are replaced by one of them which conforms to all others. It is an error if no such instance exists, or if C is not a trait (§5.3). 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 classes 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 classes of $S\#T$ are the base classes of T in B seen from the prefix type S.

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 class, 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 T is the i'th type parameter of some class D, then
  - If S has a base class $D[U_1, \ldots, U_n]$, 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 classes, 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 all bindings d such that there exists a type instance of some class C among the base classes 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.

The *definition* of a type projection $S\#t$ is the member binding d of the type t in S. In that case, we also say that $S\#t$ *is defined by* d.

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 (≡) between types is the smallest congruence[2] 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` ≡ 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` ≡ p`.type`.
- Two compound types are equivalent if their component types are pairwise equivalent and their refinements are equivalent. Two refinements are equivalent if they bind the same names and the modifiers, types and bounds of every declared entity are equivalent in both refinements.
- Two method types are equivalent if they have equivalent result types, both have the same number of parameters, and corresponding parameters have equivalent types as well as the same **def** or ⇒ modifiers. Note that the names of parameters do not matter for method type equivalence.
- Two polymorphic 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 overloaded types are equivalent if for every alternative type in either type there exists an equivalent alternative type in the other.

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 $T <: U$.

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

- For every value type T, `scala.All` $<: T <:$ `scala.Any`.
- For every value type $T <:$ `scala.AnyRef` one has `scala.AllRef` $<: T$.
- 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 c conforms to any of its base-types, b.
- A type projection $T\#t$ conforms to $U\#t$ if T conforms to U.
- A parameterized type $T[T_1, \ldots, T_n]$ conforms to $T[U_1, \ldots, U_n]$ if the following three conditions hold for $i = 1, \ldots, n$.
  - If the i'th type parameter of T is declared covariant, then $T_i <: U_i$.
  - If the i'th type parameter of T is declared contravariant, then $U_i <: T_i$.
  - 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 $i = 1, \ldots, n$ and for every binding of a type or value x in R there exists a member binding of x in T subsuming it, then T conforms to the compound type T_1 **with** $\ldots$ **with** T_n $\{R\}$.
- If $T_i \equiv T_i'$ for $i = 1, \ldots, n$ and U conforms to U' then the method type $(T_1, \ldots, T_n)U$ conforms to $(T_1', \ldots, T_n')U'$.
- If, assuming $L_1' <: a_1 <: U_1', \ldots, L_n' <: a_n <: U_n'$ one has $L_i <: L_i'$ and $U_i' <: U_i$ for $i = 1, \ldots, n$, as well as $T <: T'$, then the polymorphic type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n]T$ conforms to the polymorphic type $[a_1 >: L_1' <: U_1', \ldots, a_n >: L_n' <: U_n']T'$.
- An overloaded type $T_1 \langle\mathsf{and}\rangle \ldots \langle\mathsf{and}\rangle T_n$ conforms to each of its alternative types T_i.
- A type S conforms to the overloaded type $T_1 \langle\mathsf{and}\rangle \ldots \langle\mathsf{and}\rangle T_n$ if S conforms to each alternative type T_i.

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

- A value declaration **val** x: T or value definition **val** x: T = e subsumes a value declaration **val** x: T' if $T <: T'$.
- A type alias ***type*** $t = T$ subsumes a type alias ***type*** $t = T'$ if $T \equiv T'$.
- A type declaration **type** $t >: L <: U$ subsumes a type declaration **type** $t >: L' <: U'$ if $L' <: L$ and $U <: U'$.

- A type or class definition of some type t subsumes an abstract type declaration `type t >: L <: U` if $L <: t <: U$.

The (<:) relation forms a partial order between types. The *least upper bound* or the *greatest lower bound* of a set of types is understood to be relative to that order.

Note. The least upper 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[3].

3.6 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 a type variable is the erasure of its upper bound.
- The erasure of a 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 $|T_1|$.
- The erasure of every other type is the type itself.

3.7 Implicit Conversions

The following implicit conversions are applied to expressions of method type that are used as values, rather than being applied to some arguments.

[3] The current Scala compiler limits the nesting level of parameterization in a such bounds to 10.

- A parameterless method m of type $[]T$ is converted to type T by evaluating the expression to which m is bound.
- 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 type T by determining with local type inference (§10) instance types $T_1, \ldots, T_n$ for the type variables $a_1, \ldots, a_n$ and implicitly embedding e in the type application $e[U_1, \ldots, U_n]$ (§6.5).
- An expression e of monomorphic method type $(Ts_1)\ldots(Ts_n)U$ of arity $n > 0$ which does not appear as the function part of an application is converted to a function type by implicitly embedding e in the following term, where x is a fresh variable and each ps_i is a parameter section consisting of parameters with fresh names of types Ts_i:

  $$\texttt{(}\textbf{val}\ x = e\ \texttt{;}\ (ps_1)\ldots \Rightarrow \ldots \Rightarrow (ps_n) \Rightarrow x(ps_1)\ldots(ps_n)\texttt{)}$$

  This conversion is not applicable to functions with call-by-name parameters **def** x`: T` or repeated parameters `x: T*`, (§4.5), because its result would violate the well-formedness rules for anonymous functions (§6.19). Hence, methods with such parameters always need to be applied to arguments immediately.

When used in an expression, a value of type `byte`, `char`, or `short` is always implicitly converted to a value of type `int`.

Implicit conversions can also be user-defined. This is expained in Chapter 8.

Chapter 4

Basic Declarations and Definitions

Syntax:

```
Dcl             ::=  val ValDcl
                  |  var VarDcl
                  |  def FunDcl
                  |  type TypeDcl
Def             ::=  val PatDef
                  |  var VarDef
                  |  def FunDef
                  |  type TypeDef
                  |  TmplDef
```

A *declaration* introduces names and assigns them types. It can appear as one of the statements of a class definition (§5.1) or as part of a refinement in a compound type (3.2.5).

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 a statement sequence $s_1 \ldots s_n$, if a simple name in s_i refers to an entity defined by s_j where $j \geq i$, then every non-empty statement between and including s_i and s_j must be an import clause, or a function, type, class, or object definition. It may not be a value definition, a variable definition, or an expression.

4.1 Value Declarations and Definitions

Syntax:

```
Dcl          ::=  val ValDcl
ValDcl       ::=  id {',' id} ':' Type
Def          ::=  val PatDef
PatDef       ::=  Pattern2 {',' Pattern2} [':' Type] '=' Expr
```

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. The type T may be omitted, in which case the 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. The effect of the value definition is to bind x to the value of e converted to type T.

Value definitions can alternatively have a pattern (§7.1) 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 `=>` `scala.Tuple`$n(x_1, \ldots, x_n)$`}`
val x_1 `=` $\$x$`._1`
...
val x_n `=` $\$x$`._n` `.`

Here, $\$x$ is a fresh name. The class `Tuple`n is defined for $n = 2, \ldots, 9$ in package `scala`.

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 `=> ()}`

Example 4.1.1 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 => scala.Tuple2(x, xs) }
val x = x$._1;
val xs = x$._2;
```

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

Syntax:

```
Dcl           ::=  var VarDcl
Def           ::=  var VarDef
VarDcl        ::=  id {',' id} ':' Type
VarDef        ::=  id {',' id} [':' Type] '=' Expr
                |  id {',' id} ':' 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.

A variable definition **var** x: T = _ introduces a mutable variable 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 getter and setter functions are then members of the template instead of the variable accessed by them.

Example 4.2.1 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, m: int = 0, 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 t = 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

Syntax:

```
Dcl            ::=  type TypeDcl
TypeDcl        ::=  id [>: Type] [<: Type]
```

```
Def             ::=  type TypeDef
TypeDef         ::=  id [TypeParamClause] '=' Type
```

A *type declaration* **type** t `>:` L `<:` U declares t to be an abstract type with lower bound type L and upper bound type U. If such a declaration appears as a member declaration of a type, implementations of the type may implement t with any type T for which $L <: T <: U$. Either or both bounds may be omitted. If the lower bound L is missing, the bottom type `scala.All` is assumed. If the upper bound U is missing, the top type `scala.Any` is assumed.

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`[`*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 (§4) and type parameters (§4.5) 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`[`*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.

Example 4.3.1 The following are legal type declarations and definitions:

```
type IntList = List[Integer];
type T <: Comparable[T];
type Two[a] = Tuple2[a, a];
```

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 <: AnyRef with T;          // T is abstract, may not be part of
                                  // compound type

type T >: Comparable[T.That];     // Cannot select from T.
                                  // T is a type, not a value
```

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

Example 4.3.2 The `Predef` module contains a definition which establishes `Pair` as an alias of the parameterized class `Tuple2`:

```
type Pair[+a, +b] = Tuple2[a, b];
```

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

```
new Pair[Int, Int](1, 2) .
```

4.4 Type Parameters

Syntax:

```
TypeParamClause  ::=  '[' VarTypeParam {',' VarTypeParam} ']'
VarTypeParam     ::=  ['+' | '-'] TypeParam
TypeParam        ::=  id [>: Type] [<: Type | <% Type]
```

Type parameters appear in type definitions, class definitions, and function definitions. The most general form of a type parameter is $\pm t$ `>:` L `<:` U. Here, L, and U are lower and upper bounds that constrain possible type arguments for the parameter, and $\pm$ is a *variance*, i.e. an optional prefix of either `+`, or `-`.

The names of all type parameters in a type parameter clause must be pairwise different. 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.

Example 4.4.1 Here are some well-formed type parameter clauses:

```
[s, t]
[ex <: Throwable]
[a <: Ord[b], b <: a]
[a, b, c >: a <: b]
```

The following type parameter clauses are illegal since type parameter are bounded by themselves.

```
[a >: a]
[a <: b, b <: c, c <: a]
```

Variance annotations indicate how type instances with the given type parameters vary with respect to subtyping (§3.5.2). 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`[`tps`] =` S, type parameters labeled '+' must only appear in covariant po-

sition in S whereas type parameters labeled '–' must only appear in contravariant position. Analogously, for a class definition **class** c`[`tps`](`ps`):` s **extends** t, 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 right hand side S of a type alias **type** t`[`tps`] =` S is always in invariant position.
- 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`[`$\ldots T \ldots$`]`.

Example 4.4.2 The following variance annotation is legal.

```
abstract class P[+a, +b] {
  def fst: a; def snd: b
}
```

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

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

If we make the elements of P mutable, the variance annotation becomes illegal.

```
abstract class Q[+a, +b] {
  var fst: a;           // **** error: illegal variance:
  var snd: b            // 'a', 'b' occur in invariant position.
}
```

Example 4.4.3 The following variance annotation is illegal, since *a* appears in contravariant position in the parameter of append:

```
trait Vector[+a] {
  def append(x: Vector[a]): Vector[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:

```
trait Vector[+a] {
  def append[b >: a](x: Vector[b]): Vector[b];
}
```

Example 4.4.4 Here is a case where a contravariant type parameter is useful.

```
trait 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.

4.5 Function Declarations and Definitions

Syntax:

```
Dcl                ::=  def FunDcl
FunDcl             ::=  FunSig {',' FunSig} ':' Type
Def                ::=  def FunDef
FunDef             ::=  FunSig {',' FunSig} [':' Type] '=' Expr
FunSig             ::=  id [FunTypeParamClause] {ParamClause}
FunTypeParamClause ::= '[' TypeParam {',' TypeParam} ']'
ParamClause        ::= '(' [Param {',' Param}] ')'
Param              ::=  [def] id ':' Type ['*']
```

A function declaration has the form **def** $f\,psig$: T, where f is the function's name, *psig* is its parameter signature and T is its result type. A function definition $f\,psig$: 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 [*tps*], followed by zero or more value parameter clauses $(ps_1)\ldots(ps_n)$. Such a declaration or definition introduces a value with a (possibly polymorphic) method

type whose parameter types and result type are as given.

A type parameter clause *tps* consists of one or more type declarations (§4.3), 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, which bind value parameters and associate them with their types. The scope of a formal value parameter name x is the function body, if one is given. Both type parameter names and value parameter names must be pairwise distinct.

Value parameters may be prefixed by **`def`**, e.g. **`def`** 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*.

Example 4.5.1 The declaration

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

produces the typing

```
whileLoop: (cond: [] Boolean) (stat: [] Unit) Unit
```

which indicates that both parameters of **`while`** are evaluated using call-by-name.

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 `(`$T_1, \ldots, T_n, S$`*)`U is applied to arguments $(e_1, \ldots, e_k)$ where $k \geq n$, then m is taken in that application to have type $(T_1, \ldots, T_n, S, \ldots, S)U$, with $k-n$ occurrences of type S.

Example 4.5.2 The following method definition computes the sum of a variable number of integer arguments.

```
def sum(args: int*) {
  var result = 0;
  for (val arg <- args.elements) result = result + arg;
  result
}
```

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

```
sum()
sum(1)
```

```
sum(1, 2, 3, 4, 5)
```

The type of the function body must 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 type of the function body.

For any index i let $fsig_i$ be a function signature consisting of a function name, an optional type parameter section, and zero or more parameter sections. Then a function declaration **def** $fsig_1, \ldots, fsig_n$`:` T is a shorthand for the sequence of function declarations **def** $fsig_1$`:` T`;` `...`; **def** $fsig_n$`:` T. A function definition **def** $fsig_1, \ldots, fsig_n$ `=` e is a shorthand for the sequence of function definitions **def** $fsig_1$ `=` e`;` `...`; **def** $fsig_n$ `=` e. A function definition **def** $fsig_1, \ldots, fsig_n : T$ `=` e is a shorthand for the sequence of function definitions **def** $fsig_1 : T$ `=` e`;` `...`; **def** $fsig_n : T$ `=` e.

4.6 Overloaded Definitions

An overloaded definition is a set of $n > 1$ value or function definitions in the same statement sequence that define the same name, binding it to types $T_1, \ldots, T_n$, respectively. The individual definitions are called *alternatives*. Overloaded definitions may only appear in the statement sequence of a template. Alternatives always need to specify the type of the defined entity completely. It is an error if the types of two alternatives T_i and T_j have the same erasure (§3.6).

4.7 Import Clauses

Syntax:

```
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 (§3.1) and I is an import expression. The import expression determines a set of names of members of p which are made available without qualification. The most general form of an import expression is a list of *import selectors*

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

for $n \geq 0$, where the final wildcard '_' may be absent. It makes available each 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 members z of p other than

$x_1, \ldots, x_n$ 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 a 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.

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** $p_1.I_1$; ...; **import** $p_n.I_n$.

Example 4.7.1 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

Syntax:

```
TmplDef          ::=  ([case] class | trait) ClassDef
                   |  [case] object ObjectDef
```

Classes (§5.2) and objects (§5.4) are both defined in terms of *templates*.

5.1 Templates

Syntax:

```
Template         ::=  Constr {‘with’ Constr} [TemplateBody]
TemplateBody     ::=  ‘{’ [TemplateStat {‘;’ TemplateStat}] ‘}’
```

A template defines the type signature, behavior and initial state of a class of objects or of a single object. Templates form part of instance creation expressions, class definitions, and object definitions. A template sc **with** mc_1 **with** ... **with** mc_n $\{stats\}$ consists of a constructor invocation sc which defines the template's *superclass*, constructor invocations $mc_1, \ldots, mc_n$ ($n \geq 0$), which define the template's *mixin classes*, and a statement sequence *stats* which contains additional member definitions for the template. Superclass and mixin classes together are called the *parent classes* of a template. They must be pairwise different. The superclass of a template must be a subtype of the superclass of each mixin class. The *least proper supertype* of a template is the class type or compound type (§3.2.5) consisting of the its parent classes.

Member definitions define new members or overwrite members in the parent classes. If the template forms part of a class definition, the statement part *stats* may also contain declarations of abstract members.

Inheriting from Java Types. A template may have a Java class as its superclass and Java interfaces as its mixin classes. On the other hand, it is not permitted to have a Java class as a mixin class, or a Java interface as a superclass.

5.1.1 Constructor Invocations

Syntax:

```
Constr  ::=  StableId [TypeArgs] [‘(’ [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 $x.c(args)$, where x is a stable identifier (§3.1), c is a type name which either designates a class or defines an alias type for one, and *args* is an argument list, which matches one of the constructors of that class. The prefix ‘$x.$’ can be omitted. The argument list `(`*args*`)` can also be omitted, in which case an empty argument list `()` is implicitly added.

5.1.2 Base Classes

For every template, class type and constructor invocation we define two sets of class types: the *base classes* and *mixin base classes*. Their definitions are as follows.

The *mixin base classes* of a template sc **with** mc_1 **with** ... **with** mc_n {*stats*} are the reduced union (§3.4) of the base classes of all mixins mc_i. The mixin base classes of a class type C are the mixin base classes of the template augmented by C itself. The mixin base classes of a constructor invocation of type T are the mixin base classes of class T.

The *base classes* of a template consist are the reduced union of the base classes of its superclass and the template's mixin base classes. The base classes of class `scala.Any` consist of just the class itself. The base classes of some other class type C are the base classes of the template represented by C augmented by C itself. The base classes of a constructor invocation of type T are the base classes of T.

The notions of mixin base classes and base classes are extended from classes to arbitrary types following the definitions of §3.4.

Example 5.1.1 Consider the following class definitions:

```
class A;
class B extends A;
trait C extends A;
class D extends A;
class E extends B with C with D;
class F extends B with D with E;
```

The mixin base classes and base classes of classes `A`–`F` are given in the following table:

	Mixin base classes	Base classes
A	A	A, ScalaObject, AnyRef, Any
B	B	B, A, ScalaObject, AnyRef, Any
C	C	C, A, ScalaObject, AnyRef, Any
D	D	D, A, ScalaObject, AnyRef, Any
E	C, D, E	E, B, C, D, A, ScalaObject, AnyRef, Any
F	C, D, E, F	F, B, D, E, C, A, ScalaObject, AnyRef, Any

Note that `D` is inherited twice by `F`, once directly, the other time indirectly through `E`. This is permitted, since `D` is a trait.

5.1.3 Evaluation

The evaluation of a template or constructor invocation depends on whether the template defines an object or is a superclass of a constructed object, or whether it is used as a mixin for a defined object. In the second case, the evaluation of a template used as a mixin depends on an *actual superclass*, which is known at the point where the template is used in a definition of an object, but not at the point where it is defined. The actual superclass is used in the determination of the meaning of **`super`** (§6.3).

We therefore define two notions of template evaluation: (Plain) evaluation (as a defining template or superclass) and mixin evaluation with a given superclass *sc*. These notions are defined for templates and constructor invocations as follows.

A *mixin evaluation with superclass sc* of a template sc' **`with`** mc_1 **`with`** mc_n `{`*stats*`}` consists of mixin evaluations with superclass *sc* of the mixin constructor invocations $mc_1, \ldots, mc_n$ in the order they are given, followed by an evaluation of the statement sequence *stats*. Within *stats* the actual superclass refers to *sc*. A mixin evaluation with superclass *sc* of a class constructor invocation `ci` consists of an evaluation of the constructor function and its arguments in the order they are given, followed by a mixin evaluation with superclass *sc* of the template represented by the constructor invocation.

An *evaluation* of a template *sc* **`with`** mc_1 **`with`** mc_n **`with`** `(`*stats*`)` consists of an evaluation of the superclass constructor invocation *sc*, followed by a mixin evaluation with superclass *sc* of the template. An evaluation of a class constructor invocation `ci` consists of an evaluation of the constructor function and its arguments in the order they are given, followed by an evaluation of the template represented by the constructor invocation.

5.1.4 Template Members

The object resulting from evaluation of a template has directly bound members and inherited members. Members can be abstract or concrete. For a template T these categories are defined as follows.

1. A *directly bound* member of T is an entity introduced by a member definition or declaration in T's statement sequence. The member is called *abstract* if it is introduced by a declaration, *concrete* otherwise.
2. A *concrete inherited* member of T is a non-private, concrete member of one of T's parent classes, except if a member with the same name is already directly bound in T or the member is mixin-overridden in T. A member m of T's superclass is *mixin-overridden* in T if there is a concrete member of a mixin base class of T which either overrides m itself or overrides a member named m of a base class of T's superclass.
3. An *abstract inherited* member of T is a non-private, abstract member of one of T's parent classes P_i, except if the template has a directly bound or concrete inherited member with the same name, or the template has an abstract member inherited from a parent class P_j where $j > i$.

It is an error if a template has more than one member with the same name.

Example 5.1.2 Consider the class definitions

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

Then class `C` has a directly bound abstract member `h`. It inherits member `f` from class `B` and member `g` from class `A`.

5.1.5 Overriding

A template member M that has the same name as a non-private member M' of a base class (and that belongs to the same namespace) is said to *override* that member. In this case the binding of the overriding member M must subsume (§3.5.2) the binding of the overridden member M'. Furthermore, the overridden definition may not be a class definition. Method definitions may only override other method definitions (or the methods implicitly defined by a variable definition). They may not override value definitions. Finally, the following restrictions on modifiers apply to M and M':

- M' must not be labeled **`final`**.
- M must not be labeled **`private`**.

- 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**.
- If M' is labeled **abstract** and **override**, and M' is a member of the static superclass of the class containing the definition of M, then M must also be labeled **abstract** and **override**.

Example 5.1.3 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 trait 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 }
```

5.1.6 Modifiers

Syntax:

```
Modifier        ::=  LocalModifier
                  |  private
                  |  protected
                  |  override
LocalModifier   ::=  abstract
                  |  final
                  |  sealed
```

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 repeatedly. 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 in a template. Private members can be accessed only from within the template that defines them. Private members are not inherited by subclasses and they may not override definitions in parent classes. **private** may not be applied to abstract members, and it may not be combined in one modifier list with **protected**, **final** or **override**.
- The **protected** modifier applies to class member definitions. Protected members can be accessed from within the template of the defining class as well as

in all templates that have the defining class as a base class. A protected identifier x may be used as a member name in a selection $r.x$ only if r is one of the reserved words **this** and **super**, or if r's type conforms to a type-instance of the class which contains the access.

- The **override** modifier applies to class member definitions. It is mandatory for member definitions that override some other concrete member definition in a super- or mixin-class. If an **override** modifier is given, there must be at least one overridden member definition.

  The **override** modifier has an additional significance when combined with the **abstract** modifier. That modifier combination is only allowed for members of abstract classes. A member labeled **abstract** and **override** must override some member of the superclass of the class containing the definition.

  We call a member of a template *incomplete* if it is either abstract (i.e. defined by a declaration), or it is labeled **abstract** and **override** and it overrides an incomplete member of the template's superclass.

  Note that the **abstract override** modifier combination does not influence the concept whether a member is concrete or abstract. A member for which only a declaration is given is abstract, whereas a member for which a full definition is given is concrete.

- The **abstract** modifier is used in class definitions. It is mandatory if the class has incomplete members. Abstract classes cannot be instantiated (§6.7) with a constructor invocation unless followed by mixin constructors or statements which override all incomplete members of the class.

  The **abstract** modifier can also be used in conjunction with **override** for class member definitions. In that case the meaning of 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 redundant for them, too. **final** may not be applied to incomplete members, and it may not be combined in one modifier list with **private** or **sealed**.

- The **sealed** modifier applies to class definitions. A **sealed** class may not be inherited, except if either

  – the inheriting template is nested within the definition of the sealed class itself, or

  – the inheriting template belongs to a class or object definition which forms part of the same statement sequence as the definition of the sealed class.

Example 5.1.4 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 = 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:

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

5.1.7 Attributes

Syntax:

```
AttributeClause ::=  '[' Attribute {',' Attribute} ']'
Attribute       ::=  Constr
```

Attributes associate meta-information with definitions. A simple attribute clause has the form $[C]$ or $[C(a_1, \ldots, a_n)]$. Here, c is a constructor of a class C, which must comform to the class `scala.Attribute`. All given constructor arguments $a_1, \ldots, a_n$ must be constant expressions. An attribute clause applies to the first definition or declaration following it. More than one attribute clause may precede a definition and declaration. The order in which these clauses are given does not matter. It is also possible to combine several attributres separated by commas in one clause. Such a combined clause $[A_1, \ldots, A_n]$ is equivalent to a set of clauses $[A_1] \ldots [A_n]$.

The meaning of attribute clauses is implementation-dependent. On the Java platform, the following attributes 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>;
```

5.2 Class Definitions

Syntax:

```
ClassSig {',' ClassSig} [':' SimpleType] ClassTemplate ClassSig ::=
id [TypeParamClause] [ParamClause] ClassTemplate ::= extends
Template | TemplateBody |
```

The most general form of class definition is **class** *c*`[`*tps*`](`*ps*`):` *s* **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 `[`*tps*`]` may be omitted. A class with a type parameter section is called *polymorphic*, otherwise it is called *monomorphic*.

ps is a formal value parameter clause for the *primary constructor* of the class. The scope of a formal value parameter includes the template *t*. However, a formal value parameter may not form part of the types of any of the parent classes or members of *t*. It is illegal to define two formal value parameters with the same name. The formal parameter section `(`*ps*`)` may be omitted, in which case an empty parameter section `()` is assumed.

s is the *self type* of the class. Inside the class, the type of **this** is assumed to be *s*. The self type must conform to the self types of all classes which are inherited by the template *t*. The self type declaration '`:`*s*' may be omitted, in which case the self type of the class is assumed to be equal to *c*`[`*tps*`]`.

t is a template (§5.1) of the form

$$sc \ \textbf{with} \ mc_1 \ \textbf{with} \ \ldots \ \textbf{with} \ mc_n \ \{ \ stats \ \} \qquad (n \geq 0)$$

which defines the base classes, behavior and initial state of objects of the class. The extends clause **extends** *sc* can be omitted, in which case

extends `scala.AnyRef` is assumed. The class body `{`*stats*`}` may also be omitted, in which case the empty body `{}` is assumed.

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

For any index *i* let $csig_i$ be a class signature consisting of a class name and optional type parameter and value parameter sections. Let *ct* be a class template. Then a class definition **class** $csig_1, \ldots, csig_n$ *ct* is a shorthand for the sequence of class definitions **class** $csig_1$ *ct*`;` `...`; **class** $csig_n$ *ct*. A class definition **class** $csig_1, \ldots, csig_n : T$ *ct* is a shorthand for the sequence of class definitions **class** $csig_1 : T$ *ct*`;` `...`; **class** $csig_n : T$ *ct*.

5.2.1 Constructor Definitions

Syntax:

```
FunDef          ::=  this ParamClause'=' ConstrExpr
ConstrExpr      ::=  this ArgumentExprs
                  |  '{' this ArgumentExprs {';' BlockStat} '}'
```

A class may have additional constructors besides the primary constructor. These are defined by constructor definitions of the form **def this**`(`*ps*`) =` *e*. Such a definition introduces an additional constructor for the enclosing class, with parameters as given in the formal parameter list *ps*, and whose evaluation is defined by the constructor expression *e*. The scope of each formal parameter is the constructor expression *e*. A constructor expression is either a self constructor invocation **this**`(`*args*`)` or a block which begins with a self constructor invocation. Neither the signature, nor the self constructor invocation of a constructor definition may refer to `this`, or refer to value parameters or members of the enclosing class by simple name.

If there are auxiliary constructors of a class *C*, they define together with *C*'s primary constructor an overloaded constructor value. The usual rules for overloading resolution (§4.6) apply for constructor invocations of *C*, including the self constructor invocations in the constructor expressions themselves. 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). The type of a constructor expression must be always so that a generic instance of the class is constructed. I.e., if the class in question has name *C* and type parameters `[`*tps*`]`, then each constructor must construct an instance of *C*`[`*tps*`]`; it is not permitted to instantiate formal type parameters.

Example 5.2.1 Consider the class definition

```
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 an overloaded constructor of type

```
[a](): LinkedList[a]   ⟨and⟩
[a](x: a): LinkedList[a]    ⟨and⟩
[a](x: a, xs: LinkedList[a]): LinkedList[a]  .
```

The second constructor alternative constructs an singleton list, while the third one constructs a list with a given head and tail.

5.2.2 Case Classes

Syntax:

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

If a class definition is prefixed with **case**, the class is said to be a *case class*. The primary constructor of a case class may be used in a constructor pattern (§7.1). The following four restrictions ensure efficient pattern matching for case classes.

1. None of the base classes of a case class may be a case class.
2. No type may have two different case classes among its base types.
3. A case class may not inherit indirectly from a **sealed** class. That is, if a base class b of a case class c is marked **sealed**, then b must be a parent class of c.
4. The primary constructor of a case class may not have any call-by-name parameters (§4.5).

A case class definition of c`[`tps`](`ps`)` with type parameters tps and value parameters ps implicitly generates a function definition for a *case class factory* together with the class definition itself:

```
def c[tps](ps): s = new c[tps](ps)
```

(Here, s is the self type of class c. If a type parameter section is missing in the class, it is also missing in the factory definition).

Also implicitly defined are accessor member definitions in the class that return its value parameters. Every binding $x : T$ in the parameter section leads to a value definition of x that defines x to be an alias of the parameter.

Every case class implicitly overrides some method definitions of class `scala.AnyRef` (§12.1) 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 belong to the same class and have equal (with respect to `equals`) primary constructor arguments.
>
> Method `hashCode: ()int` computes a hash-code depending on the data structure in a way which maps equal (with respect to `equals`) values to equal hash-codes.
>
> Method `toString: ()String` returns a string representation which contains the name of the class and its primary constructor arguments.

Example 5.2.2 Here is the definition of abstract syntax for lambda calculus:

```
class Expr;
case class
  Var       (x: String)             extends Expr,
  Apply     (f: Expr, e: Expr)      extends Expr,
  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, the only classes permitted to extend `Expr` are those which are nested

inside `Expr`, or which appear in the same statement sequence as the definition of `Expr`.

5.3 Traits

Syntax:

```
TmplDef          ::=  trait ClassDef
```

A class definition which starts with the reserved word **`trait`** instead of **`class`** defines a trait. A trait is a specific instance of an abstract class, so the **`abstract`** modifier is redundant for it. The trait definition must satisfy the following four restrictions.

1. There are no value parameters in the trait's primary constructor, nor are there secondary constructors.
2. All mixin base classes of the trait are traits.
3. All parent class constructors of the trait are primary constructors with empty value parameter lists.
4. All non-empty statements in the trait's template are either imports or pure definitions.

A *pure* definition can be evaluated without any side effect. Function, type, class, or object definitions are always pure. A value definition is pure if its right-hand side expression is pure. A secondary constructor definition is pure if its right-hand side consists only Pure expressions are paths, literals, and typed expressions $e : T$ where e is pure.

These restrictions ensure that the evaluation of the mixin constructor of a trait has no effect. Therefore, traits may appear several times in the base classes of a template, whereas other classes cannot.

Example 5.3.1 The following trait class defines the property of being ordered, i.e. comparable to objects of some type. It contains an abstract method < and default implementations of the other comparison operators <=, >, and >=.

```
trait Ord[t <: Ord[t]]: 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;
}
```

5.4 Object Definitions

Syntax:

```
ObjectDef       ::=  id {',' id} [':' SimpleType] ClassTemplate
```

An object definition defines a single object of a new class. Its most general form is **object** m: s **extends** t. Here,

m is the name of the object to be defined.

s is the *self type* of the object. References to m are assumed to have type s. Furthermore, inside the template t, the type of **this** is also assumed to be s. The type of the anonymous class defined by t must conform to s and s must conform to the self types of all classes which are inherited by t. The self type declaration ': s' may be omitted, in which case the self type is assumed to be equal to the anonymous class defined by t.

t is a template (§5.1) of the form

sc **with** mc_1 **with** ... **with** mc_n { *stats* }

which defines the base classes, behavior and initial state of m. The extends clause **extends** sc can be omitted, in which case **extends** `scala.AnyRef` is assumed. The class body {*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 a class definition and a value definition that creates an object of the class:

```
final class m$cls: s extends t;
final val m: s = new m$cls;
```

(The **final** modifiers are omitted if the definition occurs as part of a block. The class name m`$cls` is not accessible for user programs.)

There are however two differences between an object definition and a pair of class and value definitions such as the one given above. First, object definitions may appear as top-level definitions in a compilation unit, whereas value definitions may not. Second, the module defined by an object definition is instantiated lazily. The **new** m`$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.

Example 5.4.1 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.

Let ct be a class template. Then an object definition **object** $x_1, \ldots, x_n$ ct is a shorthand for the sequence of object definitions **object** x_1 ct; ...; **object** x_n ct. An object definition **object** $x_1, \ldots, x_n : T$ ct is a shorthand for the sequence of object definitions **object** $x_1 : T$ ct; ...; **object** $x_n : T$ ct.

Chapter 6

Expressions

Syntax:

```
Expr            ::=  [Bindings '=>'] Expr
                  |  Expr1
Expr1           ::=  if '(' Expr ')' Expr [[';'] else Expr]
                  |  try '{' block '}' [catch Expr] [finally Expr]
                  |  while '(' Expr ')' Expr
                  |  do Expr [';'] while '(' Expr ')'
                  |  for '(' Enumerators ')' (do | yield) Expr
                  |  return [Expr]
                  |  throw Expr
                  |  [SimpleExpr '.'] id '=' Expr
                  |  SimpleExpr ArgumentExprs '=' Expr
                  |  PostfixExpr [':' Type1]
PostfixExpr     ::=  InfixExpr [id]
InfixExpr       ::=  PrefixExpr
                  |  InfixExpr id PrefixExpr
PrefixExpr      ::=  ['-' | '+' | '~' | '!'] SimpleExpr
SimpleExpr      ::=  Literal
                  |  Path
                  |  '(' [Expr] ')'
                  |  BlockExpr
                  |  new Template
                  |  SimpleExpr '.' id
                  |  SimpleExpr TypeArgs
                  |  SimpleExpr ArgumentExprs
                  |  XmlExpr
ArgumentExprs   ::=  '(' [Exprs] ')'
                  |  BlockExpr
BlockExpr       ::=  '{' CaseClause {CaseClause} '}'
                  |  '{' Block '}'
```

```
Block           ::=  {BlockStat ';'} [ResultExpr]
ResultExpr      ::=  Expr1
                  |  Bindings '=>' Block
Exprs           ::=  Expr {',' Expr}
```

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

The typing of expressions is often relative to some *expected type*. 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.

6.1 Literals

Syntax:

```
SimpleExpr    ::=  Literal
Literal       ::=  intLit
                |  floatLit
                |  charLit
                |  stringLit
                |  symbolLit
                |  true
                |  false
                |  null
```

Typing and evaluation of numeric, character, and string literals are generally as in Java. An integer literal denotes an integer number. Its type is normally `int`. However, if the expected type *pt* of the 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 expression's type is *pt*. A floating point literal denotes a single-precision or double precision IEEE floating point number. A character literal denotes a Unicode character. A string literal denotes a member of `String`.

A symbol literal `'`x is a shorthand for the expression `scala.Symbol("`x`")`. If the symbol literal is followed by actual parameters, as in `'`x`(`*args*`)`, then the whole expression is taken to be a shorthand for `scala.Symbol("`x`",` *args*`)`.

The boolean truth values are denoted by the reserved words **`true`** and **`false`**. The type of these expressions is `boolean`, and their evaluation is immediate.

The **`null`** literal is of type `scala.AllRef`. It denotes a reference value which refers to a special "null' object, which implements methods in class `scala.AnyRef` as follows:

- `eq(`x`)`, `==(`x`)`, `equals(`x`)` return **`true`** iff their argument x is also the "null" object.
- `isInstanceOf[`T`]` always returns **`false`**.

- `asInstanceOf[`T`]` returns the "null" object itself if T conforms to `scala.AnyRef`, and throws a `NullPointerException` otherwise.
- `toString()` returns the string "null".

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

6.2 Designators

Syntax:

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

A designator refers to a named term. It can be a *simple name* or a *selection*. If r is a stable identifier of type T, the selection $r.x$ refers to the term member 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 typing rules for blocks implies that in that case x's type may not refer to any abstract type member of e.

The expected type of a designator's prefix is always missing. The type of a designator is normally the type of the entity it refers to. However, if the designator is a path (§3.1) p, its type is p`.type`, provided the expression's expected type is a singleton type, or p occurs as the prefix of a selection or type selection.

The selection $e.x$ is evaluated by first evaluating the qualifier expression e. The selection's result is then the value to which the selector identifier is bound in the object resulting from evaluation of e.

6.3 This and Super

Syntax:

```
SimpleExpr    ::=  [id '.'] this
                |  [id '.'] super ['[' id ']'] '.' 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 an instance creation expression, the type of **`this`** is the type of that template. 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 inner-

most such definition. If the expression's expected type is a singleton 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 in a template refers to the definition of m in the actual superclass (§5.1.2) of the template. A reference C`.super.`m refers to the definition of m in the actual superclass of the innermost enclosing class or object definition named C which encloses the reference. The definition m referred to via `super` or C`.super` must be concrete, or the template containing the reference must have an incomplete (§5.1.6) member m' which overrides m.

The `super` prefix may be followed by a mixin qualifier `[`M`]`, as in C`.super[`M`].`x. This is called a *mixin super reference*. In this case, the reference is to the member of x in the (first) mixin class of C whose simple name is M. That member may not be abstract.

Example 6.3.1 Consider the following class definitions

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

Then we have:

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

Note that the `superB` function returns different results depending on whether `B` is used as defining class or as a mixin class.

Example 6.3.2 Consider the following class definitions:

```
class Shape {
  override def equals(other: Any) = ...;
  ...
}
trait Bordered extends Shape {
  val thickness: int;
  override def equals(other: Any) = other match {
    case that: Bordered =>
```

```
      super equals other && this.thickness == that.thickness
    case _ => false
  }
  ...
}
trait Colored extends Shape {
  val color: Color;
  override def equals(other: Any) = other match {
    case that: Colored =>
      super equals other && this.color == that.color
    case _ => false
  }
  ...
}
```

Both definitions of `equals` are combined in the class below.

```
trait BorderedColoredShape extends Shape with Bordered with Colored {
  override def equals(other: Any) =
    super[Bordered].equals(that) && super[Colored].equals(that)
}
```

6.4 Function Applications

Syntax:

```
SimpleExpr    ::=  SimpleExpr ArgumentExprs
```

An application f`(`e_1`, ..., `e_n`)` applies the function f to the argument expressions $e_1, \ldots, e_n$. If f has a method type `(`T_1`, ..., `T_n`)U`, the type of each argument expression e_i must conform to the corresponding parameter type T_i. If f has some value type, the application is taken to be equivalent to f`.apply(`e_1`, ..., `e_n`)`, i.e. the application of an `apply` method defined by f.

Evaluation of f`(`e_1`, ..., `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 **def**-parameter with a parameterless method type `[]`T is treated specially. In this case, the corresponding actual argument expression 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 the actual argument ex-

pression. In other words, the evaluation order for **def**-parameters is *call-by-name* whereas the evaluation order for normal parameters is *call-by-value*.

6.5 Type Applications

Syntax:

```
SimpleExpr    ::=  SimpleExpr '[' Types ']'
```

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 corresponding bounds L_i and U_i. That is, for each $i = 1, \ldots, n$, we must have $L_i\sigma <: T_i <: U_i\sigma$, where σ is the substitution $[a_1 := T_1, \ldots, a_n := T_n]$. The type of the application is $S\sigma$.

The function part e may also have some value type. In this case 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 (§10) can infer best type parameters for a polymorphic functions from the types of the actual function arguments and the expected result type.

6.6 References to Overloaded Bindings

If a name f referenced in an identifier or selection is overloaded (§4.6), the context of the reference has to identify a unique alternative of the overloaded binding. The way this is done depends on whether or not f is used as a function. Let $\mathscr{A}$ be the set of all type alternatives of f.

Assume first that f appears as a function in an application, as in f`(`*args*`)`. If there is precisely one alternative in $\mathscr{A}$ which is a (possibly polymorphic) method type whose arity matches the number of arguments given, that alternative is chosen.

Otherwise, let *Ts* be the vector of types obtained by typing each argument with a missing expected type. One determines first the set of applicable alternatives. A method type alternative is *applicable* if each type in *Ts* is compatible with the corresponding formal parameter type in the alternative, and, if the expected type is defined, the method's result type is compatible to it. A polymorphic method type is applicable if local type inference can determine type arguments so that the instantiated method type is applicable.

Here, a type T is *compatible* to a type U if T conforms to U after applying implicit conversions (§3.7).

Let $\mathscr{B}$ be the set of applicable alternatives. It is an error if $\mathscr{B}$ is empty. Otherwise,

one chooses the *most specific* alternative among the alternatives in $\mathscr{B}$, according to the following definition of being "more specific".

- A method type `(`*Ts*`)`*U* is more specific than some other type *S* if *S* is applicable to arguments `(`*ps*`)` of types *Ts*.
- A polymorphic method type `[`a_1 `>:` L_1 `<:` $U_1, \ldots, a_n$ `>:` L_n `<:` U_n`]T` is more specific than some other type *S* if *T* is more specific than *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.
- Any other type is always more specific than a parameterized method type or a polymorphic type.

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

Assume next that f appears as a function in a type application, as in f`[`*targs*`]`. Then we choose an alternative in $\mathscr{A}$ which takes the same number of type parameters as there are type arguments in *targs*. It is an error if no such alternative exists, or if it is not unique.

Assume finally that f 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 unique alternative in $\mathscr{B}$ which is more specific than all other alternatives in $\mathscr{B}$.

Example 6.6.1 Consider the following definitions:

```
class A extends B {}
def f(x: B, y: B) = ...
def f(x: A, y: B) = ...
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) = ...
```

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

6.7 Instance Creation Expressions

Syntax:

```
SimpleExpr     ::=  new Template
```

A simple instance creation expression is of the form **new** c where c is a constructor invocation (§5.1.1). Let T be the type of c. Then T must denote a (a type instance of) a non-abstract subclass of `scala.AnyRef` which conforms to its self type (§5.2). 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's self type (which might be less specific than T).

A general instance creation expression is of the form

new sc **with** mc_1 **with** ... **with** mc_n {*stats*}

where $n \geq 0$, sc as well as $mc_1, \ldots, mc_n$ are constructor invocations (of types $S, T_1, \ldots, T_n$, say) and *stats* is a statement sequence containing initializer statements and member definitions (§5.1.4). The type of such an instance creation expression is then the compound type S **with** T_1 **with** ... **with** T_n $\{R\}$, where $\{R\}$ is a refinement (§3.2.5) which declares exactly those members of *stats* that override a member of S or $T_1, \ldots, T_n$. For this type to be well-formed, R may not reference types defined in *stats* which do not themselves form part of R.

The instance creation expression is evaluated by creating a fresh object, which is initialized by evaluating the expression template.

Example 6.7.1 Consider the class

```
abstract class C {
  type T; val x: T; def f(x: T): AnyRef
}
```

and the instance creation expression

```
C { type T = Int; val x: T = 1; def f(x: T): T = y; val y: T = 2 }
```

Then the created object's type is:

```
C { type T = Int; val x: T; def f(x: T): T }
```

The value y is missing from the type, since y does not override a member of C.

6.8 Blocks

Syntax:

```
BlockExpr   ::=  '{' Block '}'
Block       ::=  [{BlockStat ';'} 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 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 missing.

The type of a block $s_1\,;\ \ldots;\ s_n\,;\ e$ is usually the type of e. That type must be equivalent to a type which does not refer to an entity defined locally in the block. If this condition is violated, but a fully defined expected type is given, the type of the block is instead assumed to be the expected 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.

Example 6.8.1 Written in isolation, the block

```
{ class C extends B {...} ; new C }
```

is illegal, since its type refers to class C, which is defined locally in the block.

However, when used in a definition such as

```
val x: B = { class C extends B {...} ; new C }
```

the block is well-formed, since the problematic type C can be replaced by the expected type B.

6.9 Prefix, Infix, and Postfix Operations

Syntax:

```
PostfixExpr     ::=  InfixExpr [id]
InfixExpr       ::=  PrefixExpr
                   |  InfixExpr id PrefixExpr
PrefixExpr      ::=  ['-' | '+' | '!' | '~'] SimpleExpr
```

Expressions can be constructed from operands and operators. A prefix operation $op\,e$ consists of a prefix operator op, which must be one of the identifiers '+', '-', '!', or '~', and a simple expression e. The expression is equivalent to the postfix method application $e.op$.

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)`.

An infix or postfix 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.

(all letters)
|
^
&
< >
= !
:
+ -
* / %
(all other special characters)

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

The *associativity* of an operator is determined by the operator's last character. Operators ending with 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 $e_0\ op_1\ e_1\ op_2 \ldots op_n\ e_n$ with operators $op_1, \ldots, op_n$ of the same precedence, then all these operators must have the same associativity. If all operators are left-associative, the sequence is interpreted as $(\ldots(e_0\ op_1\ e_1)\ op_2 \ldots)\ op_n\ e_n$. Otherwise, if all operators are right-associative, the sequence is interpreted as $e_0\ op_1\ (e_1\ op_2\ (\ldots op_n\ e_n)\ldots)$.
- Postfix operators always have lower precedence than infix operators. E.g. $e_1\ op_1\ e_2\ op_2$ is always equivalent to $(e_1\ op_1\ e_2)\ op_2$.

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

6.10 Typed Expressions

Syntax:

```
Expr1              ::=  PostfixExpr [':' Type1]
```

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.

Example 6.10.1 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
```

6.11 Assignments

Syntax:

```
Expr1       ::=  Designator '=' Expr
              |  SimpleExpr 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 $f(args) = e$ with a function application to the left of the "=' operator is interpreted as f.`update(`$args$`,` e`)`, i.e. the invocation of an `update` function defined by f.

Example 6.11.1 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.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 + xs(i)(k) * yss(k)(j);
        k = k + 1
      }
      zss(i)(j) = acc;
      j = j + 1
```

```
    }
    i = 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.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.apply(i).apply(k) * yss.apply(k).apply(j);
        k = k + 1
      }
      zss.apply(i).update(j, acc);
      j = j + 1
    }
    i = i + 1
  }
  zss
}
```

6.12 Conditional Expressions

Syntax:

```
Expr1          ::=  if '(' Expr ')' Expr [[';'] 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 least upper bound of the types of e_1 and e_2. 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** (). The type of this expression is unit and the then-part *e*2 is also expected to conform to type unit.

6.13 While Loop Expressions

Syntax:

```
Expr1          ::=  while '(' Expr ')' 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(def c: boolean)(def s: unit): unit  =
  if (c) { s ; while(c)(s) } else {}
```

Example 6.13.1 The loop

```
while (x != 0) { y = y + 1/x ; x = x - 1 }
```

Is equivalent to the application

```
whileLoop (x != 0) { y = y + 1/x ; x = x - 1 }
```

Note that this application will never produce a division-by-zero error at run-time, since the expression (y = 1/x) will be evaluated in the body of **while** only if the condition parameter is false.

6.14 Do Loop Expressions

Syntax:

```
Expr1          ::=  do Expr [';'] 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.

6.15 Comprehensions

Syntax:

```
Expr1          ::=  for '(' Enumerators ')' [yield] Expr
Enumerator     ::=  Generator {';' Enumerator}
Enumerator     ::=  Generator
                 |  Expr
Generator      ::=  val Pattern1 '<-' Expr
```

A comprehension **for** (*enums*) **yield** *e* evaluates expression *e* for each binding generated by the enumerators *enums*. Enumerators start with a generator, which can be followed by further generators or filters. A *generator* **val** *p* <- *e* produces bindings from an expression *e* which is matched in some way against pattern *p*. A *filter* is an expressions which restricts enumerated bindings. The precise meaning of generators and filters is defined by translation to invocations of four methods: `map`, `filter`, `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 **val** *p* <- *e*, where *p* is not a pattern variable, is replaced by

val *p* <- *e*.filter { **case** *p* => **true**; **case** _ => **false** }

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

- A generator **val** *p* <- *e* followed by a filter *f* is translated to a single generator **val** *p* <- *e*.filter($x_1, \ldots, x_n$ => *f*) where $x_1, \ldots, x_n$ are the free variables of *p*.
- A for-comprehension **for** (**val** *p* <- *e*) **yield** e' is translated to *e*.map { **case** *p* => e' }.
- A for-comprehension **for** (**val** *p* <- *e*) e' is translated to *e*.foreach { **case** *p* => e' }.
- A for-comprehension

  for (**val** *p* <- *e*; **val** p' <- e' ...) **yield** e'' ,

  where ... is a (possibly empty) sequence of generators or filters, is translated to

  e.flatmap { **case** *p* => **for** (**val** p' <- e' ...) **yield** e'' } .

- A for-comprehension

  for (**val** *p* <- *e*; **val** p' <- e' ...) e'' .

  where ... is a (possibly empty) sequence of generators or filters, is translated to

  e.foreach { **case** *p* => **for** (**val** p' <- e' ...) e'' } .

Example 6.15.1 the following code produces all pairs of numbers between 1 and $n-1$ whose sums are prime.

```
for  { val i <- range(1, n);
       val j <- range(1, i);
       isPrime(i+j)
} yield Pair (i, j)
```

The for-comprehension is translated to:

```
range(1, n)
  .flatMap {
     case i => range(1, i)
       .filter { j => isPrime(i+j) }
       .map { case j => Pair(i, j) } }
```

Example 6.15.2 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 (val i <- Array.range(0, xss(0).length)) yield
    Array(for (val 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 (val Pair(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 Example 6.11.1.

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

The code above makes use of the fact that `map`, `flatmap`, `filter`, and `foreach` are defined for members of class `scala.Array`.

6.16 Return Expressions

Syntax:

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

A return expression **return** *e* must occur inside the body of some enclosing named method or function f. This function 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.All`.

6.17 Throw Expressions

Syntax:

```
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 (§6.18) 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.All`.

6.18 Try Expressions

Syntax:

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

A try expression **try** { *b* } **catch** *e* evaluates the block *b*. If evaluation of *b* does not cause an exception to be thrown, the result of *b* is returned. Otherwise the *handler e* is applied to the thrown exception. Let *pt* be the expected type of the try expression. The block *b* is expected to conform to *pt*. The handler *e* is expected conform to type `scala.PartialFunction[scala.Throwable,` *pt*`]`. The type of the try expression is the least upper bound of the type of *b* and the result type of *e*.

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.

6.19 Anonymous Functions

Syntax:

```
Expr1          ::=  Bindings '=>' Expr
ResultExpr     ::=  Bindings '=>' Block
Bindings       ::=  '(' Binding {',' Binding ')'
                  |  id [':' Type1]
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 missing. The type of the anonymous function is `scala.Function`n`[`$S_1, \ldots, S_n$`,` T`]`, where T is the 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

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

In the case of a single formal parameter, `(`x`:` T`) =>` e and `(`x`) =>` e can be abbreviated to x`:` T `=>` e, and x `=>` e, respectively.

Example 6.19.1 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 = count + 1; count } // The function which takes an
                                        // empty parameter list (),
                                        // increments a non-local variable
                                        // 'count' and returns the new value.
```

6.20 Statements

Syntax:

```
BlockStat    ::=  Import
                |  Def
                |  {LocalModifier} TmplDef
                |  Expr
                |
TemplateStat ::=  Import
                |  {AttributeClause} {Modifier} Def
                |  {AttributeClause} {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 modifiers allowed in block-local definitions are modifiers **`abstract`**, **`final`**, or **`sealed`** preceding a class or object definition.

With the exception of overloaded definitions (§4.6), a statement sequence making up a block or template may not contain two definitions or declarations that bind the same name in the same namespace. Evaluation of a statement sequence entails evaluation of the statements in the order they are written.

Chapter 7

Pattern Matching

7.1 Patterns

Syntax:

```
Pattern       ::=  SimplePattern {Id SimplePattern}
                |  varid ‘:’ Type
                |  ‘_’ ‘:’ Type
SimplePattern ::=  varid
                |  ‘_’
                |  literal
                |  StableId {‘(’ [Patterns] ‘)’}
                |  XmlPattern
Patterns      ::=  Pattern {‘,’ Pattern}
```

For clarity, this section deals with a subset of the Scala pattern language. The extended Scala pattern language, which is described below, adds more flexible variable binding and regular hedge expressions.

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

A pattern is built from constants, constructors, variables and regular operators. 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.

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

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.

A *typed pattern* $x : T$ consists of a pattern variable x and a simple type T. The type T may be a class type or a compound type; it may not contain a refinement (§3.2.5). This pattern matches any value of type T and binds the variable name to that value. T must conform to the pattern's expected type. The type of x is T.

A *pattern literal* l matches any value that is equal (in terms of ==) to it. It's type must conform to the expected type of the pattern.

A *named pattern constant* r is a stable identifier (§3.1). To resolve the syntactic overlap with a variable pattern, a named pattern constant may not be a simple name starting with a lower-case letter. The type of r must conform to the expected type of the pattern. The pattern matches any value v such that `$r$ == $v$` (§12.1).

A *constructor pattern* $c(p_1)\ldots(p_n)$ where $n \geq 0$ consists of an identifier c, followed by component patterns $p_1, \ldots, p_n$. The constructor c is either a simple name or a qualified name $r.id$ where r is a stable identifier. It refers to a (possibly overloaded) function which has one alternative of result type `class C`, and which may not have other overloaded alternatives with a class constructor type as result type. Furthermore, the respective type parameters and value parameters of (said alternative of) c and of the primary constructor function of class C must be the same, after renaming corresponding type parameter names. If C is monomorphic, then `C` must conform to the expected type of the pattern, and the formal parameter types of C's primary constructor are taken as the expected types of the component patterns $p_1, \ldots, p_n$. If C is polymorphic, then there must be a unique type application instance of it such 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 component pattern p_i matches the corresponding value v_i.

An *infix operation pattern* `p id p'` is a shorthand for the constructor pattern `id_class(p)(p')`. The precedence and associativity of operators in patterns is the same as in expressions (§6.9).

Example 7.1.1 Some examples of patterns are:

1. The pattern `ex: IOException` matches all instances of class `IOException`, binding variable `ex` to the instance.
2. 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.
3. The pattern `x :: y :: xs` matches lists of length ≥ 2, binding `x` to the lists's first element, `y` to the list's second element, and `xs` to the remainder.

7.1.1 Regular Pattern Matching

Syntax:

```
Pattern         ::=  Pattern1 { '|' Pattern1 }
Pattern1        ::=  varid ':' Type
                  |  '_' ':' Type
                  |  Pattern2
Pattern2        ::=  [varid '@'] Pattern3
Pattern3        ::=  SimplePattern [ '*' | '?' | '+' ]
                  |  SimplePattern { id' SimplePattern }
SimplePattern   ::=  '_'
                  |  varid
                  |  Literal
                  |  StableId [ '(' [Patterns] ')' ]
                  |  '(' [Patterns] ')'
Patterns        ::=  Pattern {',' Pattern}
id'             ::=  id  but not  '*' | '?' | '+' | '@' | '|'
```

We distinguish between tree patterns and hedge patterns (hedges are ordered sequences of trees). A *tree pattern* describes a set of matching trees (like above). A *hedge pattern* describes a set of matching hedges. Both kinds of patterns may contain *variable bindings* which serve to extract constituents of a tree or hedge.

The type of a patterns and the expected types of variables within patterns are determined by the context and the structure of the patterns. The last case ensures that a variable bound to a hedge pattern will have a sequence type.

The following patterns are added:

A *hedge pattern* $p_1, \ldots, p_n$ where $n \geq 0$ is a sequence of patterns separated by commas and matching the hedge described by the components. Hedge patterns may appear as arguments to constructor applications, or nested within a another hedge pattern if grouped with parentheses. Note that empty hedge patterns are allowed. The type of tree patterns that appear in a hedge pattern is the expected type as determined from the enclosing constructor. A *fixed-length argument pattern* is a special hedge pattern where where all p_i are tree patterns.

A *choice pattern* $p_1|\ldots|p_n$ is a choice among several alternatives, which may not contain variable-binding patterns. It matches every tree and every hedge matched by at least one of its alternatives. Note that the empty sequence may appear as an alternative. An *option pattern* $p?$ is an abbreviation for $(p|)$. A choice is a tree pattern if all its branches are tree patterns. In this case, all branches must conform to the expected type and the type of the choice is the least upper bound of the branches. Otherwise, its type is determined by the enclosing hedge pattern it is part of.

An *iterated pattern* $p*$ matches zero, one or more occurrences of items matched by p, where p may be either a tree pattern or a hedge pattern. p may not contain a variable-binding. A *non-empty iterated pattern* $p+$ is an abbreviation for $(p, p*)$.

The treatment of the following patterns changes with to the previous section:

A *constructor pattern* $c(p)$ consists of a simple type c followed by a pattern p. If c designates a monomorphic case class, then it must conform to the expected type of the pattern, the pattern must be a fixed length argument pattern $p_1, \ldots, p_n$ whose length corresponds to the number of arguments of c's primary constructor. The expected types of the component patterns are then taken from the formal parameter types of (said) constructor. If c designates a polymorphic case class, then there must be a unique type application instance of it such 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 $p_1, \ldots, p_n$. In both cases, the pattern matches all objects created from constructor invocations $c(v_1, \ldots, v_n)$ where each component pattern p_i matches the corresponding value v_i. If c does not designate a case class, it must be a subclass of `Seq[`T`]`. In that case p may be an arbitrary sequence pattern. Value patterns in p are expected to conform to type T, and the pattern matches all objects whose `elements()` method returns a sequence that matches p.

The pattern (p) is regarded as equivalent to the pattern p, if p is a nonempty sequence pattern. The empty tuple `()` is a shorthand for the constructor pattern `Unit`.

A *variable-binding* $x@p$ is a simple identifier x which starts with a lower case letter, together with a pattern p. It matches every item (tree or hedge) matched by p, and in addition binds it to the variable name. If p is a tree pattern of type T, the type of x is also T. If p is a hedge pattern enclosed by constructor c `<:Seq[`T`]`, then the type of x is `List[`T`]` where T is the expected type as dictated by the constructor.

Regular expressions that contain variable bindings may be ambiguous, i.e. there might be several ways to match a sequence against the pattern. In these cases, the *right-longest policy* applies: patterns that appear more to the right than others in a sequence take precedence in case of overlaps.

Example 7.1.2 Some examples of patterns are:

1. The pattern `ex: IOException` matches all instances of class `IOException`, binding variable `ex` to the instance.
2. The pattern `Pair(x, _)` matches pairs of values, binding `x` to the first component of the pair. The second component is matched with a wildcard pattern.
3. The pattern `List( 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, which may be empty.
4. The pattern `List( 1, x@(( 'a' | 'b' )+),y,_ )` matches a list that contains 1 as its first element, continues with a non-empty sequence of `'a`'s and `'b`'s, followed by two more elements. The sequence 'a's and 'b's is bound to `x`, and the next to last element is bound to `y`.

5. The pattern `List( x@( 'a'* ), 'a'+ )` matches a non-empty list of `'a'`s. Because of the shortest match policy, `x` will always be bound to the empty sequence.
6. The pattern `List( x@( 'a'+ ), 'a'* )` also matches a non-empty list of `'a'`s. Here, `x` will always be bound to the sequence containing one `'a'`

7.2 Pattern Matching Expressions

Syntax:

```
BlockExpr       ::=  '{' CaseClause {CaseClause} '}'
CaseClause      ::=  case Pattern ['if' PostfixExpr] '=>' Block
```

A pattern matching expression **case** p_1 => b_1 ... **case** p_n => b_n consists of a number $n \geq 1$ of cases. Each case consists of a (possibly guarded) pattern p_i and a block b_i. The scope of the pattern variables in p_i is the corresponding block b_i.

The expected type of a pattern matching expression must in part be defined. It must be either `scala.Function1[`T_p`,` T_r`]` or `scala.PartialFunction[`T_p`,` T_r`]`, where the argument type T_p must be fully determined, but the result type T_r may be undetermined. All patterns are typed relative to the expected type T_p (§7.1). The expected type of every block b_i is T_r. Let T_b be the least upper bound of the types of all blocks b_i. The type of the pattern matching expression is then the required type with T_r replaced by T_b (i.e. the type is either `scala.Function[`T_p`,` T_b`]` or `scala.PartialFunction[`T_p`,` T_b`]`.

When applying a pattern matching expression to a selector value, patterns are tried in sequence until one is found which matches the selector value (§7.1). 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.

Example 7.2.1 Often, pattern matching expressions are used as arguments of the `match` method, which is predefined in class `Any` (§12.1) and is implemented there by postfix function application. Here is an example:

```
def length [a] (xs: List[a]) = xs match {
  case Nil =>  0
  case x :: xs1  =>  1 + length (xs1)
}
```

In an application of `match` such as the one above, the expected type of all patterns is the type of the qualifier of `match`. In the example above, the expected type of the patterns `Nil` and `x :: xs1` would be `List[a]`, the type of `xs`.

Chapter 8

Views

Views are user-defined, implicit coercions that are automatically inserted by the compiler.

8.1 View Definition

A view definition is a normal function definition with one value parameter where the name of the defined function is `view`.

Example 8.1.1 The following defines an implicit coercion function from strings to lists of characters.

```
def view(xs: String): List[char] =
  if (xs.length() == 0) List()
  else xs.charAt(0) :: xs.substring(1);
```

8.2 View Application

View applications are inserted implicitly in two situations.

1. Around an expression e of type T, if T does not conform to the expression's expected type PT.
2. In a selection $e.m$ with e of type T, if the selector m does not denote a member of T.

In the first case, a view method `view` is searched which is applicable to e and whose result type conforms to PT. If such a method is found, the expression e is converted to `view(`e`)`.

In the second case, a view method `view` is searched which is applicable to e and whose result contains a member named m. If such a method is found, the selection $e.m$ is converted to `view(`e`).m`

8.3 Finding Views

Searching a view which is applicable to an expression e of type T is a three-step process.

1. First, the set $\mathscr{A}$ of available views is determined. $\mathscr{A}$ is the smallest set such that:

   (a) If a unary method called `view` is accessible without qualifier anywhere on the path of the program tree that leads from e to the root of the tree (describing the whole compilation unit), then that method is in the set $\mathscr{A}$. Methods are accessible without qualifier because they are locally defined in an enclosing scope, or imported into an enclosing scope, or inherited by an enclosing class.

   (b) If a unary method called `view` is a member of an object C such that there is a base class C of T with the same name as the object and defined in the same scope, then that method is in the set $\mathscr{A}$.

2. Then, among all the methods in $\mathscr{A}$ the set of all applicable views $\mathscr{B}$ is determined. A view method is applicable if it can be applied to values of type T, and another condition is satisfied which depends on the context of the view application:

   (a) If the view is a conversion to a given prototype PT, then the view's result type must conform to PT.

   (b) If the view is a conversion in a selection with member m, then the view's result type must contain a member named m.

   Note that in the determining of view applicability, we do not permit further views to be inserted. I.e. a view is applicable to an expression e of type T if it can be applied to e, without a further view conversion of e to the view's formal parameter type. Likewise, a view's result type must conform to a given prototype directly, no second view conversion is allowed.

3. It is an error if the set of applicable views $\mathscr{B}$ is empty. For non-empty $\mathscr{B}$, the view method which is most specific (§6.6) in $\mathscr{B}$ is selected. It is an error if no most specific view exists, or if it is not unique.

Example 8.3.1 Consider the following situation.

```
class A;
class B extends A;
class C;
object B {
  def view(x: B): C = ...
}
object Test with Application {
  def view(x: A): C = ...
  val x: C = new B;
}
```

For the expression `new B` there are two available views. The view defined in object `B` is available since its associated class is (a superclass of) the expression's type `B`. The view defined in object `Test` is available since it is accessible without qualification at the point of the expression `new B`. Both views are also applicable since they map values of type `B` to results of type `C`. However, the view defined in object `B` is more specific than the view defined in object `Test`. Hence, the last statement in the example above is implicitly augmented to

```
val x: C = B.view(new B)
```

8.4 View-Bounds

Syntax:

```
TypeParam       ::=  id [>: Type] [<% Type]
```

A type parameter `a` may have a view bound `a <% T` instead of a regular upper bound `a <: T`. In that case the type parameter may be instantiated to any type `S` which is convertible by application of a view method to the view bound `T`. Here, we assume there exists an always available identity view method

```
def view[a](x: a): a = x .
```

Hence, the type parameter `a` can always be instantiated to subtypes of the view bound `T`, just as if `T` was a regular upper bound.

View bounds for type parameters behave analogously to upper bounds wrt to type conformance (§3.5.2), variance checking (§4.4), and overriding (§5.1.5).

Methods or classes with view-bounded type parameters implicitly take view functions as parameters. For every view-bounded type parameter `a <% T` one adds an implicit value parameter `view: a => T`. When instantiating the type parameter `a` to some type `S`, the most specific applicable view method from type `S` to type `T` is selected, according to the rules of §8.3. This method is then passed as actual argument to the corresponding view parameter.

Implicit view parameters of a method or class are then taken as available view methods in its body.

Example 8.4.1 Consider the following definition of a trait `Comparable` and a view from strings to that trait.

```
trait Comparable[a] {
  def less(x: a): boolean
}

object StringsAreComparable {
  def view(x: String): Comparable[String] = new Comparable[String] {
    def less(y: String) = x.compareTo(y) < 0
  }
}
```

Now, define a binary tree with a method `insert` which inserts an element in the tree and a method `elements` which returns a sorted list of all elements of the tree. The tree is defined for all types of elements a that are viewable as `Comparable[a]`.

```
trait Tree[a <% Comparable[a]] {
  def insert(x: a): Tree[a] = this match {
    case Empty() => new Node(x, Empty(), Empty())
    case Node(elem, l, r) =>
      if (x == elem) this
      else if (x less elem) Node(elem, l insert x, r)
      else Node(elem, l, r insert x);
  }
  def elements: List[a] = this match {
    case Empty() => List()
    case Node(elem, l, r) =>
      l.elements ::: List(elem) ::: r.elements
  }
}
case class Empty[a <% Comparable[a]]()
  extends Tree[a];
case class Node[a <% Comparable[a]](elem: a, l: Tree[a], r: Tree[a])
  extends Tree[a];
```

Finally, define a test program which builds a tree from all command line argument strings and then prints out the elements as a sorted sequence.

```
object Test {
  import StringsAreComparable.view;

  def main(args: Array[String]) = {
    var t: Tree[String] = Empty();
```

```
    for (val s <- args) { t = t insert s }
    System.out.println(t.elements)
  }
}
```

Note that the definition `var t: Tree[String] = Empty();` is legal because at that point a view method from `String` to `Comparable[String]` has been imported and is therefore accessible without a prefix. The imported view method is passed as an implicit argument to the `Empty` constructor.

Here is the `Test` program again, this time with implicit views made visible:

```
object Test {
  import StringsAreComparable.view;

  def main(args: Array[String]) = {
    var t: Tree[String] = Empty(StringsAreComparable.view);
    for (val s <- args) { t = t insert s }
    System.out.println(t.elements)
  }
}
```

And here are the tree classes with implicit views added:

```
trait Tree[a <% Comparable[a]](view: a => Comparable[a]) {
  def insert(x: a): Tree[a] = this match {
    case Empty(_) => new Node(x, Empty(view), Empty(view))
    case Node(_, elem, l, r) =>
      if (x == elem) this
      else if (view(x) less elem) Node(view, elem, l insert x, r)
      else Node(view, elem, l, r insert x);
  }
  def elements: List[a] = this match {
    case Empty(_) => List()
    case Node(_, elem, l, r) =>
      l.elements ::: List(elem) ::: r.elements
  }
}
case class Empty[a <% Comparable[a]](view: a => Comparable[a])
  extends Tree[a];
case class Node[a <% Comparable[a]](view: a => Comparable[a],
                                    elem: a, l: Tree[a], r: Tree[a])
  extends Tree[a];
```

Note that views entail a certain run-time overhead because they need to be passed as additional arguments to view-bounded methods and classes. Furthermore, every application of a view entails the construction of an object which is often immedi-

ately discarded afterwards – see for instance with the translation of `(x less elem)` in the implementation of method `insert` above. It is expected that the latter cost can be absorbed largely or completely by compiler optimizations (which are, however, not yet implemented at the present stage).

8.5 Conditional Views

View methods might themselves have view-bounded type parameters; this allows the definition of conditional views.

Example 8.5.1 The following view makes lists comparable, *provided* the list element type is also comparable.

```
def view[a <% Comparable[a]](xs: List[a]): Comparable[List[a]] =
  new Comparable[List[a]] {
    def less (ys: List[a]): boolean =
      !ys.isEmpty
      &&
      (xs.isEmpty ||
       (xs.head less ys.head) ||
       (xs.head == ys.head) && (xs.tail less ys.tail))
  }
```

Note that the condition `(xs.head less ys.head)` invokes the `less` method of the list element type, which is unknown at the point of the definition of the view method. As usual, view-bounded type parameters are translated to implicit `view` arguments. In this case, the `view` method over lists would receive the `view` method over list elements as implicit parameter.

Chapter 9

Top-Level Definitions

Syntax:

```
CompilationUnit  ::=  [package QualId ';'] {TopStat ';'} TopStat
TopStat          ::=  {AttributeClause} {Modifier} TmplDef
                    |  Import
                    |  Packaging
                    |
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*; *stats* starting with a package clause is equivalent to a compilation unit consisting of a single packaging **package** *p* { *stats* }.

Implicitly imported into every compilation unit are, in that order : the package `java.lang`, the package `scala`, and the object `scala.Predef` (§12.4). Members of a later import in that order hide members of an earlier import.

9.1 Packagings

Syntax:

```
Packaging       ::=  package QualId '{' {TopStat ';'} TopStat '}'
```

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 ds` injects all definitions in `ds` as members into the package whose qualified name is *p*. If a definition in `ds` is labeled **private**, it is visible

only for other members in the package.

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.

Example 9.1.1 The following example will create a hello world program as function `main` of module `test.HelloWorld`.

```
package test;

object HelloWord {
  def main(args: Array[String]) = System.out.println("hello world")
}
```

Chapter 10

Local Type Inference

To be completed.

Chapter 11

XML expressions and patterns

This chapter describes the syntactic structure of XML expressions and patterns. It follows as close as possible the XML 1.0 specification [W3Cb], changes being mandated by the possibility of embedding Scala code fragments.

11.1 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 (see 1.5).

Syntax:

```
XmlExpr ::= Element {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 neither declarations, CDATA sections nor processing instructions. Entity references are not resolved at runtime.

Syntax:

```
Element       ::=   EmptyElemTag
                  |  STag Content ETag

EmptyElemTag  ::=   '<' Name {S Attribute} [S] '/>'
```

```
STag          ::=   '<' Name {S Attribute} [S] '>'
ETag          ::=   '</' Name [S] '>'
Content       ::=   [CharData] {Content1 [CharData]}
Content1      ::=   Element
                |   Reference
                |   CDSect
                |   PI
                |   Comment
                |   ScalaExpr
```

If an XML expression is a single element, its value is a runtime representation of an XML node (an instance of a subclass of `scala.xml.Node`). If the XML expression consists of more than one element, then its value is a runtime representation of a sequence of XML nodes (an instance of a subclass of `scala.Seq[scala.xml.Node]`).

If an XML expression is an entity reference, CDATA section, processing instructions or a comments, it is represented by an instance of the corresponding Scala runtime class.

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 behaviour can be changed to preserve all whitespace with a compiler option. **Syntax:**

```
Attribute     ::=   Name Eq AttValue

AttValue      ::=   '"' {CharQ | CharRef} '"'
                |   ''' {CharA | CharRef} '''
                |   ScalaExp

ScalaExpr     ::=   '{' expr '}'

CharData      ::=  { CharNoRef } without {CharNoRef}'{'CharB {CharNoRef}
                                  and without {CharNoRef}']]>'{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.

Syntax:

```
BaseChar, Char, Comment, CombiningChar, Ideographic, NameChar, S, Reference
              ::=  "as in W3C XML"

Char1         ::=  Char without '<' | '&'
CharQ         ::=  Char1 without '"'
```

```
CharA          ::=  Char1 without '''
CharB          ::=  Char1 without '{'

Name           ::=  XNameStart {NameChar}

XNameStart     ::= '_' | BaseChar | Ideographic
                   (as in W3C XML, but without ':'
```

11.2 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 (see 1.5).

Syntax:

```
XmlPattern  ::= ElementPattern {ElementPattern}
```

Well-formedness constraints of the XML specification apply.

If an XML pattern is a single element pattern, it expects the type of runtime representation of an XML tree, and matches exactly one instance of this type that has the same structure as described by the pattern. If an XML pattern consists of more than one element, then it expects the type of sequences of runtime representations of XML trees, and matches every sequence whose elements match the sequence described by the pattern.

XML patterns may contain Scala patterns(7.2).

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 behaviour can be changed to preserve all whitespace with a compiler option.

Syntax:

```
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 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.

12.1 Root Classes

The root of the Scala class 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 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 standard interfaces of these root classes is described by the following definitions.

```
package scala;
abstract class Any {

  /** Reference equality */
  final def eq(that: Any): boolean = ...
```

```
  /** Defined equality */
  def equals(that: Any): boolean = this eq that;

  /** Semantic equality between values of same type */
  final def == (that: Any): boolean  =  this equals that

  /** Semantic inequality between values of same type */
  final def != (that: Any): boolean  =  !(this == that)

  /** Hash code */
  def hashCode(): Int = ...

  /** Textual representation */
  def toString(): String = ...

  /** Type test */
  def isInstanceOf[a]: Boolean = match {
    case x: a => true
    case _ => false
  }

  /** Type cast */
  def asInstanceOf[a]: a = match {
    case x: a => x
    case _ => if (this eq null) this
              else throw new ClassCastException()
  }

  /** Pattern match */
  def match[a, b](cases: a => b): b = cases(this);
}
final class AnyVal extends Any;
class AnyRef extends Any;
trait ScalaObject extends AnyRef;
```

The type cast operation `asInstanceOf` has a special meaning (not expressed in the code above) when its type parameter is a numeric type. For any type `T <: Double`, and any numeric value `v` `v.asInstanceIf[T]` converts `v` to type `T` 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`).

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 Class `Double`

```
package scala;
abstract sealed class Double extends AnyVal {
  def + (that: Double): Double // double addition
  def - (that: Double): Double // double subtraction
  def * (that: Double): Double // double multiplication
  def / (that: Double): Double // double division
  def % (that: Double): Double // double remainder

  def == (that: Double): Boolean // double equality
  def != (that: Double): Boolean // double inequality
  def < (that: Double): Boolean  // double less
  def > (that: Double): Boolean  // double greater
  def <= (that: Double): Boolean // double less or equals
  def >= (that: Double): Boolean // double greater or equals

  def - : Double = 0.0 - this    // double negation
  def + : Double = this
}
```

12.2.2 Class `Float`

```
package scala;
abstract sealed class Float extends AnyVal {
  def coerce: Double              // convert to Double

  def + (that: Double): Double;   // double addition
  def + (that: Float): Double     // float addition
  /* analogous for -, *, /, % */

  def == (that: Double): Boolean; // double equality
  def == (that: Float): Boolean;  // float equality
  /* analogous for !=, <, >, <=, >= */

  def - : Float;                  // float negation
  def + : Float
```

```
}
```

12.2.3 Class Long

```
package scala;
abstract sealed class Long extends AnyVal  {
  def coerce: Double              // convert to Double
  def coerce: Float               // convert to Float

  def + (that: Double): Double;   // double addition
  def + (that: Float): Double;    // float addition
  def + (that: Long): Long =      // long addition
  /* analogous for -, *, /, % */

  def << (cnt: Int): Long         // long left shift
  def >> (cnt: Int): Long         // long signed right shift
  def >>> (cnt: Int): Long        // long unsigned right shift
  def & (that: Long): Long        // long bitwise and
  def | (that: Long): Long        // long bitwise or
  def ^ (that: Long): Long        // long bitwise exclusive or

  def == (that: Double): Boolean; // double equality
  def == (that: Float): Boolean;  // float equality
  def == (that: Long): Boolean    // long equality
  /* analogous for !=, <, >, <=, >= */

  def - : Long;                   // long negation
  def + : Long;                   // long identity
  def ~ : Long                    // long bitwise negation
}
```

12.2.4 Class Int

```
package scala;
abstract sealed class Int extends AnyVal {
  def coerce: Double              // convert to Double
  def coerce: Float               // convert to Float
  def coerce: Long                // convert to Long

  def + (that: Double): Double;   // double addition
  def + (that: Float): Double;    // float addtion
  def + (that: Long): Long;       // long addition
  def + (that: Int): Int;         // int addition
  /* analogous for -, *, /, % */

  def << (cnt: Int): Int;         // int left shift
```

```
  /* analogous for >>, >>> */

  def & (that: Long): Long;       // long bitwise and
  def & (that: Int): Int;         // int bitwise and
  /* analogous for |, ^ */

  def == (that: Double): Boolean; // double equality
  def == (that: Float): Boolean;  // float equality
  def == (that: Long): Boolean    // long equality
  def == (that: Int): Boolean     // int equality
  /* analogous for !=, <, >, <=, >= */

  def - : Int;                    // int negation
  def + : Int;                    // int identity
  def ~ : Int;                    // int bitwise negation
}
```

12.2.5 Class Short

```
package scala;
abstract sealed class Short extends AnyVal {
  def coerce: Double              // convert to Double
  def coerce: Float               // convert to Float
  def coerce: Long                // convert to Long
  def coerce: Int                 // convert to Int
}
```

12.2.6 Class Char

```
package scala;
abstract sealed class Char extends AnyVal {
  def coerce: Double              // convert to Double
  def coerce: Float               // convert to Float
  def coerce: Long                // convert to Long
  def coerce: Int                 // convert to Int

  def isDigit: Boolean;           // is this character a digit?
  def isLetter: Boolean;          // is this character a letter?
  def isLetterOrDigit: Boolean;   // is this character a letter or digit?
  def isWhiteSpace                // is this a whitespace character?
}
```

12.2.7 Class Short

```
package scala;
abstract sealed class Short extends AnyVal {
```

```
  def coerce: Double              // convert to Double
  def coerce: Float               // convert to Float
  def coerce: Long                // convert to Long
  def coerce: Int                 // convert to Int
  def coerce: Short               // convert to Short
}
```

12.2.8 Class Boolean

```
package scala;
abstract sealed class Boolean extends AnyVal {
  def && (def x: Boolean): Boolean; // boolean and
  def || (def x: Boolean): Boolean; // boolean or
  def &  (x: Boolean): Boolean;     // boolean strict and
  def |  (x: Boolean): Boolean      // boolean strict or

  def == (x: Boolean): Boolean      // boolean equality
  def != (x: Boolean): Boolean      // boolean inequality

  def !  (x: Boolean): Boolean      // boolean negation
}
```

12.2.9 Class Unit

```
package scala;
abstract sealed class Unit extends AnyVal;
```

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 by HTML pages elsewhere.

12.3.1 Class String

The `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 Tuplen[+a_1, ..., +a_n](_1: a_1, ..., _n: a_n) {
  def toString = "(" ++ _1 ++ "," ++ ... ++ "," ++_n ++ ")"
}
```

The implicitly imported `Predef` object (§12.4) 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;
class Functionn[-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 (§12.4) defines the name `Function` as an alias of `Function1`.

12.3.4 Class `Array`

The class of generic arrays is given as follows.

```
package scala;
class Array[a](length: int) with Function[Int, a] {
  def length: int;
  def apply(i: Int): a;
  def update(i: Int)(x: a): Unit;
}
```

12.4 The `Predef` Object

The `Predef` module defines standard functions and type aliases for Scala programs. It is always implicitly imported, so that all its defined members are available without qualification. Here is its definition for the JVM environment.

```
package scala;
object Predef {
  type byte = scala.Byte;
  type short = scala.Short;
  type char = scala.Char;
  type int = scala.Int;
  type long = scala.Long;
  type float = scala.Float;
  type double = scala.Double;
  type boolean = scala.Boolean;
  type unit = scala.Unit;

  type String = java.lang.String;
  type NullPointerException = java.lang.NullPointerException;
  type Throwable = java.lang.Throwable;
  // other aliases to be identified

  /** Abort with error message */
  def error(message: String): All = throw new Error(message);

  /** Throw an error if given assertion does not hold. */
  def assert(assertion: Boolean): Unit =
    if (!assertion) throw new Error("assertion failed");

  /** Throw an error with given message if given assertion does not hold */
  def assert(assertion: Boolean, message: Any): Unit = {
    if (!assertion) throw new Error("assertion failed: " + message);

  /** Create an array with given elements */
  def Array[A](xs: A*): Array[A] = {
    val array: Array[A] = new Array[A](xs.length);
    var i = 0;
    for (val x <- xs.elements) { array(i) = x; i = i + 1; }
    array;
  }

  /** Aliases for pairs and triples */
  type Pair[+p, +q] = Tuple2[p, q];
  def Pair[a, b](x: a, y: b) = Tuple2(x, y);
  type Triple[+a, +b, +c] = Tuple3[a, b, c];
```

```
  def Triple[a, b, c](x: a, y: b, z: c) = Tuple3(x, y, z);

  /** Alias for unary functions */
  type Function = Function1;

  /** Some standard simple functions */
  def id[a](x: a): a = x;
  def fst[a](x: a, y: Any): a = x;
  def scd[a](x: Any, y: a): a = y;
}
```

12.5 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 labelled 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( val x <- child.elements; x.label == that.name ) {
              res = x::res;
            }
            res.reverse
        }
      });
    }

  /** XPath style projection function. Returns all nodes labelled 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);

}
```

Bibliography

[GR83] Adele Goldberg and David Robson. *Smalltalk-80; The Language and Its Implementation*. Addison-Wesley, 1983. ISBN 0-201-11371-6.

[Mat01] Yukihiro Matsumoto. *Ruby in a Nutshell*. O'Reilly & Associates, nov 2001. ISBN 0-596-00214-9.

[OCRZ03] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A nominal theory of objects with dependent types. In *Proc. FOOL 10*, January 2003. `http://www.cis.upenn.edu/~bcpierce/FOOL/FOOL10.html`.

[vRD03] Guido van Rossum and Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd, sep 2003. ISBN 0-954-16178-5 `http://www.python.org/doc/current/ref/ref.html`.

[W3Ca] W3C. Document object model (DOM). `http://www.w3.org/DOM/`.

[W3Cb] W3C. Extensible markup language (XML). `http://www.w3.org/TR/REC-xml`.

Chapter A

Scala Syntax Summary

The lexical syntax of Scala is given by the following grammar in EBNF form.

```
upper          ::=  'A' | ... | 'Z' | '$' | '_'  and Unicode Lu
lower          ::=  'a' | ... | 'z'  and Unicode Ll
letter         ::=  upper | lower  and Unicode categories Lo, Lt, Nl
digit          ::=  '0' | ... | '9'
special        ::=  "all other characters in \u0020-007F and Unicode categories Sm, So
                    except parentheses ([]) and periods"

op             ::=  special {special}
varid          ::=  lower {letter | digit} ['_' {digit} [id]]
id             ::=  upper {letter | digit} ['_' {digit} [id]]
                  | varid
                  | op
                  | '\'stringLit

intLit         ::=  "as in Java"
floatLit       ::=  "as in Java"
charLit        ::=  "as in Java"
stringLit      ::=  "as in Java"
symbolLit      ::= '\'' id

comment        ::=  '/*' ''any sequence of characters'' '*/'
                  |  '//' 'any sequence of characters up to end of line''
```

The context-free syntax of Scala is given by the following EBNF grammar.

```
Literal        ::=  intLit
                  |  floatLit
                  |  charLit
                  |  stringLit
```

```
                  |  symbolLit
                  |  true
                  |  false
                  |  null

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

Type           ::=  Type1 '=>' Type
                  |  '(' [Types] ')' '=>' Type
                  |  Type1
Type1          ::=  SimpleType {with SimpleType} [Refinement]
SimpleType     ::=  SimpleType TypeArgs
                  |  SimpleType '#' id
                  |  StableId
                  |  Path '.' type
                  |  '(' Type ')'
TypeArgs       ::=  '[' Types ']'
Types          ::=  Type {',' Type}
Refinement     ::=  '{' [RefineStat {';' RefineStat}] '}'
RefineStat     ::=  Dcl
                  |  type TypeDef
                  |

Exprs          ::=  Expr {',' Expr}
Expr           ::=  Bindings '=>' Expr
                  |  Expr1
Expr1          ::=  if '(' Expr1 ')' Expr [[';'] else Expr]
                  |  try '{' Block '}' [catch Expr] [finally Expr]
                  |  do Expr [';'] while '(' Expr ')'
                  |  for '(' Enumerators ')' (do | yield) Expr
                  |  return [Expr]
                  |  throw Expr
                  |  [SimpleExpr '.'] id '=' Expr
                  |  SimpleExpr ArgumentExprs '=' Expr
                  |  PostfixExpr [':' Type1]
PostfixExpr    ::=  InfixExpr [id]
InfixExpr      ::=  PrefixExpr
                  |  InfixExpr id PrefixExpr
PrefixExpr     ::=  ['-' | '+' | '~' | '!'] SimpleExpr
SimpleExpr     ::=  Literal
                  |  Path
                  |  '(' [Expr] ')'
```

```
                  |  BlockExpr
                  |  new Template
                  |  SimpleExpr '.' id
                  |  SimpleExpr TypeArgs
                  |  SimpleExpr ArgumentExprs
                  |  XmlExpr
ArgumentExprs    ::=  '(' [Exprs] ')'
                  |  BlockExpr
BlockExpr        ::=  '{' CaseClause {CaseClause} '}'
                  |  '{' Block '}'
Block            ::=  {BlockStat ';'} [ResultExpr]
BlockStat        ::=  Import
                  |  Def
                  |  {LocalModifier} TmplDef
                  |  Expr1
                  |
ResultExpr       ::=  Expr1
                  |  Bindings '=>' Block

Enumerators      ::=  Generator {';' Enumerator}
Enumerator       ::=  Generator
                  |  Expr
Generator        ::=  val Pattern1 '<-' Expr

CaseClause       ::=  case Pattern ['if' PostfixExpr] '=>' Block

Constr           ::=  StableId [TypeArgs] ['(' [Exprs] ')']
SimpleConstr     ::=  Id [TypeArgs] ['(' [Exprs] ')']

Pattern          ::=  Pattern1 { '|' Pattern1 }
Pattern1         ::=  varid ':' Type
                  |  '_' ':' Type
                  |  Pattern2
Pattern2         ::=  [varid '@'] Pattern3
Pattern3         ::=  SimplePattern [ '*' | '?' | '+' ]
                  |  SimplePattern { id SimplePattern }
SimplePattern    ::=  '_'
                  |  varid
                  |  Literal
                  |  StableId [ '(' [Patterns] ')' ]
                  |  '(' [Patterns] ')'
                  |  XmlPattern
Patterns         ::=  Pattern {',' Pattern}

TypeParamClause ::=  '[' VarTypeParam {',' VarTypeParam} ']'
FunTypeParamClause ::=  '[' TypeParam {',' TypeParam} ']'
```

```
VarTypeParam    ::=  ['+' | '-'] TypeParam
TypeParam       ::=  id [>: Type] [<: Type | <% Type]
ParamClause     ::=  '(' [Param {',' Param}] ')'
Param           ::=  [def] id ':' Type ['*']
Bindings        ::=  id [':' Type1]
                  |  '(' Binding {',' Binding ')'
Binding         ::=  id [':' Type]

Modifier        ::=  LocalModifier
                  |  private
                  |  protected
                  |  override
LocalModifier   ::=  abstract
                  |  final
                  |  sealed

AttributeClause ::=  '[' Attribute {',' Attribute} ']'
Attribute       ::=  Constr

Template        ::=  Constr {'with' Constr} [TemplateBody]
TemplateBody    ::=  '{' [TemplateStat {';' TemplateStat}] '}'
TemplateStat    ::=   Import
                  |  {AttributeClause} {Modifier} Def
                  |  {AttributeClause} {Modifier} Dcl
                  |  Expr
                  |

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

Dcl             ::=  val ValDcl
                  |  var VarDcl
                  |  def FunDcl
                  |  type TypeDcl

ValDcl          ::=  id {',' id} ':' Type
VarDcl          ::=  id {',' id} ':' Type
FunDcl          ::=  FunSig {',' FunSig} ':' Type
FunSig          ::=  id [FunTypeParamClause] {ParamClause}
TypeDcl         ::=  id [>: Type] [<: Type]

Def             ::=  val PatDef
                  |  var VarDef
```

```
                  |  def FunDef
                  |  type TypeDef
                  |  TmplDef
PatDef          ::=  Pattern2 {',' Pattern2} [':' Type] '=' Expr
VarDef          ::=  id {',' id} [':' Type] '=' Expr
                  |  id {',' id} ':' Type '=' '_'
FunDef          ::=  FunSig {',' FunSig} ':' Type '=' Expr
                  |  this ParamClause '=' ConstrExpr
TypeDef         ::=  id [TypeParamClause] '=' Type

TmplDef         ::=  ([case] class | trait) ClassDef
                  |  [case] object ObjectDef
ClassDef        ::=  ClassSig {',' ClassSig} [':' SimpleType] ClassTemplate
ClassSig        ::=  id [TypeParamClause] [ParamClause]
ObjectDef       ::=  id {',' id} [':' SimpleType] ClassTemplate

ClassTemplate   ::=  extends Template
                  |  TemplateBody
                  |
ConstrExpr      ::=  this ArgumentExprs
                  |  '{' this ArgumentExprs {';' BlockStat} '}'

CompilationUnit ::=  [package QualId ';'] {TopStat ';'} TopStat
TopStat         ::=  {AttributeClause} {Modifier} TmplDef
                  |  Import
                  |  Packaging
                  |
Packaging       ::=  package QualId '{' {TopStat ';'} TopStat '}'
QualId          ::=  id {'.' id}
```

Chapter B

Implementation Status

The present Scala compiler does not yet implement all of the Scala specification. Its currently existing omissions and deviations are listed below. We are working on a refined implementation that addresses these issues.

1. Unicode support is still limited. At present we only permit Unicode encodings \uXXXX in strings and backquote-enclosed identifiers. To define or access a Unicode identifier, you need to put it in backquotes and use the \uXXXX encoding.
2. The unicode operator "⇒" (§1.1) is not yet recognized; you need to use the two character ASCII equivalent "=>" instead.
3. The current implementation does not yet support run-time types. All types are erased (§3.6) during compilation. This means that the following operations give potentially wrong results.

   - Type tests and type casts to parameterized types. Here it is only tested that a value is an instance of the given top-level type constructor.
   - Type tests and type casts to type parameters and abstract types. Here it is only tested that a value is an instance of the type parameter's upper bound.
   - Polymorphic array creation. If `t` is a type variable or abstract type, then **`new`** `Array[t]` will yield an array of the upper bound of `t`.

4. Return expressions are not yet permitted inside an anonymous function or inside a call-by-name argument (i.e. a function argument corresponding to a **`def`** parameter).
5. Members of the empty package (§9.1) cannot yet be accessed from other source files. Hence, all library classes and objects have to be in some package.

6. At present, auxiliary constructors (§5.2.1) are only permitted for monomorphic classes.
7. The `Array` class supports as yet only a restricted set of operations as given in §12.3.4. It is planned to extend that interface. In particular, arrays will implement the `scala.Seq` trait as well as the methods needed to support for-comprehensions.
8. At present, all classes used as mixins must be accessible to the Scala compiler in source form.