Programming in Scala

DRAFT
May 21, 2007

Martin Odersky

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.

By contrast, tree transformation is the natural domain of functional languages. Their algebraic data types, pattern matching and higher-order functions make these languages ideal for the task. It's no wonder, then, that specialized languages for transforming XML data such as XSLT are functional.

Another reason why functional language constructs are attractive for web-services is that mutable state is problematic in this setting. Components with mutable state are harder to replicate or to restore after a failure. Data with mutable state is harder to cache than immutable data. Functional language constructs make it relatively easy to construct components without mutable state.

Many web services are constructed by combining different languages. For instance, a service might use XSLT to handle document transformation, XQuery for database access, and Java for the "business logic". The downside of this approach is that the necessary amount of cross-language glue can make applications cumbersome to write, verify, and maintain. A particular problem is that cross-language interfaces are usually not statically typed. Hence, the benefits of a static type system are missing where they are needed most – at the join points of components written in different paradigms.

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 to 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

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

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 [OCRZ03a].

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] 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.

Acknowledgments. Many people have contributed to the definition and implementation of the Scala language and to parts of this book. First of all, I would like to thank the Scala team at EPFL consisting of Philippe Altherr, Vincent Cremet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. They put a lot of effort in the Scala compiler, tools, and documentation and have contributed in an essential way to the specification of the Scala language through many observations, clever suggestions, and discussions. Members of the team have also contributed examples in this book, as well as parts of the specification. Phil Bagwell, Gilad Bracha, Erik Ernst, Erik Mejer, Benjamin Pierce, Enno Runne, and Phil Wadler have given very useful feedback on the Scala design.

The documentation ows a great debt to Abelson's and Sussman's wonderful book "Structure and Interpretation of Computer Programs"[ASS96]. I have adapted several of their examples and exercises in the "Scala By Example" part of this book. Of course, the working language has in each case been changed from Scheme to Scala. Furthermore, the examples make use of Scala's object-oriented constructs where appropriate.

II Scala by Example

Scala is a programming language that fuses elements from object-oriented and functional programming. This part introduces Scala in an informal way, through a sequence of examples.

Chapters 1 and 2 highlight some of the features that make Scala interesting. The following chapters introduce the language constructs of Scala in a more thorough way, starting with simple expressions and functions, and working up through objects and classes, lists and streams, mutable state, pattern matching to more complete examples that show interesting programming techniques. The present informal exposition is complemented by the Scala Language Reference Manual which specifies Scala in a more detailed and precise way.

Chapter 1

A First Example

As a first example, here is an implementation of Quicksort in Scala.

```
def sort(xs: Array[int]) {
  def swap(i: int, j: int) {
    val t = xs(i); xs(i) = xs(j); xs(j) = t
  }
  def sort1(l: int, r: int) {
    val pivot = xs((l + r) / 2)
    var i = l; var j = r
    while (i <= j) {
      while (xs(i) < pivot) { i = i + 1 }
      while (xs(j) > pivot) { j = j - 1 }
      if (i <= j) {
        swap(i, j)
        i = i + 1
        j = j - 1
      }
    }
    if (l < j) sort1(l, j)
    if (j < r) sort1(i, r)
  }
  sort1(0, xs.length - 1)
}
```

The implementation looks quite similar to what one would write in Java or C. We use the same operators and similar control structures. There are also some minor syntactical differences. In particular:

- Definitions start with a reserved word. Function definitions start with **`def`**, variable definitions start with **`var`** and definitions of values (i.e. read only variables) start with **`val`**.

- The declared type of a symbol is given after the symbol and a colon. The declared type can often be omitted, because the compiler can infer it from the context.
- Array types are written `Array[T]` rather than `T[]`, and array selections are written `a(i)` rather than `a[i]`.
- Functions can be nested inside other functions. Nested functions can access parameters and local variables of enclosing functions. For instance, the name of the array `a` is visible in functions `swap` and `sort1`, and therefore need not be passed as a parameter to them.

So far, Scala looks like a fairly conventional language with some syntactic peculiarities. In fact it is possible to write programs in a conventional imperative or object-oriented style. This is important because it is one of the things that makes it easy to combine Scala components with components written in mainstream languages such as Java, C# or Visual Basic.

However, it is also possible to write programs in a style which looks completely different. Here is Quicksort again, this time written in functional style.

```
def sort(xs: Array[int]): Array[int] =
  if (xs.length <= 1) xs
  else {
    val pivot = xs(xs.length / 2)
    Array.concat(
      sort(xs filter (pivot >)),
           xs filter (pivot ==),
      sort(xs filter (pivot <)))
  }
```

The functional program captures the essence of the quicksort algorithm in a concise way:

- If the array is empty or consists of a single element, it is already sorted, so return it immediately.
- If the array is not empty, pick an an element in the middle of it as a pivot.
- Partition the array into two sub-arrays containing elements that are less than, respectively greater than the pivot element, and a third array which contains elements equal to pivot.
- Sort the first two sub-arrays by a recursive invocation of the sort function.[1]
- The result is obtained by appending the three sub-arrays together.

[1]This is not quite what the imperative algorithm does; the latter partitions the array into two sub-arrays containing elements less than or greater or equal to pivot.

Both the imperative and the functional implementation have the same asymptotic complexity – $O(N\ log(N))$ in the average case and $O(N^2)$ in the worst case. But where the imperative implementation operates in place by modifying the argument array, the functional implementation returns a new sorted array and leaves the argument array unchanged. The functional implementation thus requires more transient memory than the imperative one.

The functional implementation makes it look like Scala is a language that's specialized for functional operations on arrays. In fact, it is not; all of the operations used in the example are simple library methods of a *sequence* class `Seq[t]` which is part of the standard Scala library, and which itself is implemented in Scala. Because arrays are instances of `Seq` all sequence methods are available for them.

In particular, there is the method `filter` which takes as argument a *predicate function* that maps array elements to boolean values. The result of `filter` is an array consisting of all the elements of the original array for which the given predicate function is true. The `filter` method of an object of type `Array[t]` thus has the signature

```
def filter(p: t => boolean): Array[t]
```

Here, `t => boolean` is the type of functions that take an element of type `t` and return a `boolean`. Functions like `filter` that take another function as argument or return one as result are called *higher-order* functions.

Scala does not distinguish between identifiers and operator names. An identifier can be either a sequence of letters and digits which begins with a letter, or it can be a sequence of special characters, such as "+", "*", or ":". Any identfier can be used as an infix operator in Scala. The binary operation $E\ op\ E'$ is always interpreted as the method call $E.op(E')$. This holds also for binary infix operators which start with a letter. Hence, the expression `xs filter (pivot >)` is equivalent to the method call `xs.filter(pivot >)`.

In the quicksort program, `filter` is applied three times to an anonymous function argument. The first argument, `pivot >`, represents a function that takes an argument x and returns the value `pivot >` x. Another way to write this function which makes the missing argument explicit is `x => pivot > x`. The function is anonymous, i.e. it is not defined with a name. The type of the `x` parameter is omitted because a Scala compiler can infer it automatically from the context where the function is used. To summarize, `xs.filter(pivot >)` returns a list consisting of all elements of the list xs that are smaller than `pivot`.

Looking again in detail at the first, imperative implementation of Quicksort, we find that many of the language constructs used in the second solution are also present, albeit in a disguised form.

For instance, "standard" binary operators such as `+`, `-`, or `<` are not treated in any special way. Like `append`, they are methods of their left operand. Consequently, the expression `i + 1` is regarded as the invocation `i.+(1)` of the `+` method of the integer value `x`. Of course, a compiler is free (if it is moderately smart, even expected) to recognize the special case of calling the `+` method over integer arguments and to generate efficient inline code for it.

For efficiency and better error diagnostics the **`while`** loop is a primitive construct in Scala. But in principle, it could have just as well been a predefined function. Here is a possible implementation of it:

```
def While (p: => boolean) (s: => unit): unit =
      if (p) { s ; While(p)(s) }
```

The `While` function takes as first parameter a test function, which takes no parameters and yields a boolean value. As second parameter it takes a command function which also takes no parameters and yields a result of type `unit`. `While` invokes the command function as long as the test function yields true.

Scala's `unit` type roughly corresponds to `void` in Java; it is used whenever a function does not return an interesting result. In fact, because Scala is an expression-oriented language, every function returns some result. If no explicit return expression is given, the value `{}`, which is pronounced "unit", is assumed. This value is of type `unit`. Unit-returning functions are also called *procedures*. Here's a more "expression-oriented" formulation of the `swap` function in the first implementation of quicksort, which makes this explicit:

```
def swap(i: int, j: int): unit = {
  val t = xs(i); xs(i) = xs(j); xs(j) = t
  {}
}
```

The result value of this function is simply its last expression – a **`return`** keyword is not necessary. Note that functions returning an explicit value always need an "=" before their body or defining expression.

Chapter 2

Programming with Actors and Messages

Here's an example that shows an application area for which Scala is particularly well suited. Consider the task of implementing an electronic auction service. We use an Erlang-style actor process model to implement the participants of the auction. Actors are objects to which messages are sent. Every actor has a "mailbox" of its incoming messages which is represented as a queue. It can work sequentially through the messages in its mailbox, or search for messages matching some pattern.

For every traded item there is an auctioneer actor that publishes information about the traded item, that accepts offers from clients and that communicates with the seller and winning bidder to close the transaction. We present an overview of a simple implementation here.

As a first step, we define the messages that are exchanged during an auction. There are two abstract base classes `AuctionMessage` for messages from clients to the auction service, and `AuctionReply` for replies from the service to the clients. For both base classes there exists a number of cases, which are defined in Figure 2.1.

For each base class, there are a number of *case classes* which define the format of particular messages in the class. These messages might well be ultimately mapped to small XML documents. We expect automatic tools to exist that convert between XML documents and internal data structures like the ones defined above.

Figure 2.2 presents a Scala implementation of a class `Auction` for auction actors that coordinate the bidding on one item. Objects of this class are created by indicating

- a seller actor which needs to be notified when the auction is over,
- a minimal bid,
- the date when the auction is to be closed.

The behavior of the actor is defined by its `act` method. That method repeatedly

```
import scala.actors.Actor

abstract class AuctionMessage
case class Offer(bid: int, client: Actor)  extends AuctionMessage
case class Inquire(client: Actor)          extends AuctionMessage

abstract class AuctionReply
case class  Status(asked: int, expire: Date) extends AuctionReply
case object BestOffer                        extends AuctionReply
case class  BeatenOffer(maxBid: int)         extends AuctionReply
case class  AuctionConcluded(seller: Actor, client: Actor)
                                             extends AuctionReply
case object AuctionFailed                    extends AuctionReply
case object AuctionOver                      extends AuctionReply
```

Listing 2.1: Message Classes for an Auction Service

selects (using `receiveWithin`) a message and reacts to it, until the auction is closed, which is signaled by a `TIMEOUT` message. Before finally stopping, it stays active for another period determined by the `timeToShutdown` constant and replies to further offers that the auction is closed.

Here are some further explanations of the constructs used in this program:

- The `receiveWithin` method of class `Actor` takes as parameters a time span given in milliseconds and a function that processes messages in the mailbox. The function is given by a sequence of cases that each specify a pattern and an action to perform for messages matching the pattern. The `receiveWithin` method selects the first message in the mailbox which matches one of these patterns and applies the corresponding action to it.
- The last case of `receiveWithin` is guarded by a `TIMEOUT` pattern. If no other messages are received in the meantime, this pattern is triggered after the time span which is passed as argument to the enclosing `receiveWithin` method. `TIMEOUT` is a special message, which is triggered by the `Actor` implementation itself.
- Reply messages are sent using syntax of the form `destination ! SomeMessage`. `!` is used here as a binary operator with an actor and a message as arguments. This is equivalent in Scala to the method call `destination.!(SomeMessage)`, i.e. the invocation of the `!` method of the destination actor with the given message as parameter.

The preceding discussion gave a flavor of distributed programming in Scala. It might seem that Scala has a rich set of language constructs that support actor processes, message sending and receiving, programming with timeouts, etc. In fact, the

```
class Auction(seller: Actor, minBid: int, closing: Date) extends Actor {
  val timeToShutdown = 36000000; // msec
  val bidIncrement = 10
  def act() {
    var maxBid = minBid - bidIncrement
    var maxBidder: Actor = null
    var running = true
    while (running) {
      receiveWithin ((closing.getTime() - new Date().getTime())) {
        case Offer(bid, client) =>
          if (bid >= maxBid + bidIncrement) {
            if (maxBid >= minBid) maxBidder ! BeatenOffer(bid)
            maxBid = bid; maxBidder = client; client ! BestOffer
          } else {
            client ! BeatenOffer(maxBid)
          }
        case Inquire(client) =>
          client ! Status(maxBid, closing)
        case TIMEOUT =>
          if (maxBid >= minBid) {
            val reply = AuctionConcluded(seller, maxBidder)
            maxBidder ! reply; seller ! reply
          } else {
            seller ! AuctionFailed
          }
          receiveWithin(timeToShutdown) {
            case Offer(_, client) => client ! AuctionOver
            case TIMEOUT => running = false
          }
      }
    }
  }
}
```

Listing 2.2: Implementation of an Auction Service

opposite is true. All the constructs discussed above are offered as methods in the library class `Actor`. That class is itself implemented in Scala, based on the underlying thread model of the host language (e.g. Java, or .NET). The implementation of all features of class `Actor` used here is given in Section 16.11.

The advantages of the library-based approach are relative simplicity of the core language and flexibility for library designers. Because the core language need not specify details of high-level process communication, it can be kept simpler and more general. Because the particular model of messages in a mailbox is a library module, it can be freely modified if a different model is needed in some applications. The approach requires however that the core language is expressive enough to provide the necessary language abstractions in a convenient way. Scala has been designed with this in mind; one of its major design goals was that it should be flexible enough to act as a convenient host language for domain specific languages implemented by library modules. For instance, the actor communication constructs presented above can be regarded as one such domain specific language, which conceptually extends the Scala core.

Chapter 3

Expressions and Simple Functions

The previous examples gave an impression of what can be done with Scala. We now introduce its constructs one by one in a more systematic fashion. We start with the smallest level, expressions and functions.

3.1 Expressions And Simple Functions

A Scala system comes with an interpreter which can be seen as a fancy calculator. A user interacts with the calculator by typing in expressions. The calculator returns the evaluation results and their types. Example:

```
> 87 + 145
232: scala.Int

> 5 + 2 * 3
11: scala.Int

> "hello" + " world!"
hello world: scala.String
```

It is also possible to name a sub-expression and use the name instead of the expression afterwards:

```
> def scale = 5
def scale: int

> 7 * scale
35: scala.Int

> def pi = 3.141592653589793
def pi: scala.Double
```

```
> def radius = 10
def radius: scala.Int

> 2 * pi * radius
62.83185307179586: scala.Double
```

Definitions start with the reserved word **def**; they introduce a name which stands for the expression following the = sign. The interpreter will answer with the introduced name and its type.

Executing a definition such as `def x = e` will not evaluate the expression `e`. Instead `e` is evaluated whenever `x` is used. Alternatively, Scala offers a value definition `val x = e`, which does evaluate the right-hand-side `e` as part of the evaluation of the definition. If `x` is then used subsequently, it is immediately replaced by the pre-computed value of `e`, so that the expression need not be evaluated again.

How are expressions evaluated? An expression consisting of operators and operands is evaluated by repeatedly applying the following simplification steps.

- pick the left-most operation
- evaluate its operands
- apply the operator to the operand values.

A name defined by **def** is evaluated by replacing the name by the (unevaluated) definition's right hand side. A name defined by **val** is evaluated by replacing the name by the value of the definitions's right-hand side. The evaluation process stops once we have reached a value. A value is some data item such as a string, a number, an array, or a list.

Example 3.1.1 Here is an evaluation of an arithmetic expression.

```
    (2 * pi) * radius
→   (2 * 3.141592653589793) * radius
→   6.283185307179586 * radius
→   6.283185307179586 * 10
→   62.83185307179586
```

The process of stepwise simplification of expressions to values is called *reduction*.

3.2 Parameters

Using **def**, one can also define functions with parameters. Example:

```
> def square(x: double) = x * x
def (x: double): scala.Double

> square(2)
4.0: scala.Double

> square(5 + 3)
64.0: scala.Double

> square(square(4))
256.0: scala.Double

> def sumOfSquares(x: double, y: double) = square(x) + square(y)
def sumOfSquares(scala.Double,scala.Double): scala.Double

> sumOfSquares(3, 2 + 2)
25.0: scala.Double
```

Function parameters follow the function name and are always enclosed in parentheses. Every parameter comes with a type, which is indicated following the parameter name and a colon. At the present time, we only need basic numeric types such as the type `scala.Double` of double precision numbers. Scala defines *type aliases* for some standard types, so we can write numeric types as in Java. For instance `double` is a type alias of `scala.Double` and `int` is a type alias for `scala.Int`.

Functions with parameters are evaluated analogously to operators in expressions. First, the arguments of the function are evaluated (in left-to-right order). Then, the function application is replaced by the function's right hand side, and at the same time all formal parameters of the function are replaced by their corresponding actual arguments.

Example 3.2.1

```
    sumOfSquares(3, 2+2)
→   sumOfSquares(3, 4)
→   square(3) + square(4)
→   3 * 3 + square(4)
→   9 + square(4)
→   9 + 4 * 4
→   9 + 16
→   25
```

The example shows that the interpreter reduces function arguments to values before rewriting the function application. One could instead have chosen to apply the function to unreduced arguments. This would have yielded the following reduction sequence:

```
    sumOfSquares(3, 2+2)
→   square(3) + square(2+2)
→   3 * 3 + square(2+2)
→   9 + square(2+2)
→   9 + (2+2) * (2+2)
→   9 + 4 * (2+2)
→   9 + 4 * 4
→   9 + 16
→   25
```

The second evaluation order is known as *call-by-name*, whereas the first one is known as *call-by-value*. For expressions that use only pure functions and that therefore can be reduced with the substitution model, both schemes yield the same final values.

Call-by-value has the advantage that it avoids repeated evaluation of arguments. Call-by-name has the advantage that it avoids evaluation of arguments when the parameter is not used at all by the function. Call-by-value is usually more efficient than call-by-name, but a call-by-value evaluation might loop where a call-by-name evaluation would terminate. Consider:

```
> def loop: int = loop
def loop: scala.Int

> def first(x: int, y: int) = x
def first(x: scala.Int, y: scala.Int): scala.Int
```

Then `first(1, loop)` reduces with call-by-name to `1`, whereas the same term reduces with call-by-value repeatedly to itself, hence evaluation does not terminate.

```
    first(1, loop)
→   first(1, loop)
→   first(1, loop)
→   ...
```

Scala uses call-by-value by default, but it switches to call-by-name evaluation if the parameter type is preceded by =>.

Example 3.2.2

```
> def constOne(x: int, y: => int) = 1
constOne(x: scala.Int, y: => scala.Int): scala.Int

> constOne(1, loop)
1: scala.Int

> constOne(loop, 2)                  // gives an infinite loop.
```

```
^C
```

3.3 Conditional Expressions

Scala's **if-else** lets one choose between two alternatives. Its syntax is like Java's **if-else**. But where Java's **if-else** can be used only as an alternative of statements, Scala allows the same syntax to choose between two expressions. That's why Scala's **if-else** serves also as a substitute for Java's conditional expression `... ? ... : ...`.

Example 3.3.1

```
> def abs(x: double) = if (x >= 0) x else -x
abs(x: double): double
```

Scala's boolean expressions are similar to Java's; they are formed from the constants **true** and **false**, comparison operators, boolean negation `!` and the boolean operators && and `||`.

3.4 Example: Square Roots by Newton's Method

We now illustrate the language elements introduced so far in the construction of a more interesting program. The task is to write a function

```
def sqrt(x: double): double = ...
```

which computes the square root of x.

A common way to compute square roots is by Newton's method of successive approximations. One starts with an initial guess y (say: `y = 1`). One then repeatedly improves the current guess y by taking the average of y and x/y. As an example, the next three columns indicate the guess y, the quotient x/y, and their average for the first approximations of $\sqrt{2}$.

```
1           2/1 = 2               1.5
1.5         2/1.5 = 1.3333        1.4167
1.4167      2/1.4167 = 1.4118     1.4142
1.4142      ...                   ...
```

y $\qquad$ x/y $\qquad$ $(y + x/y)/2$

One can implement this algorithm in Scala by a set of small functions, which each represent one of the elements of the algorithm.

We first define a function for iterating from a guess to the result:

```
def sqrtIter(guess: double, x: double): double =
  if (isGoodEnough(guess, x)) guess
  else sqrtIter(improve(guess, x), x)
```

Note that `sqrtIter` calls itself recursively. Loops in imperative programs can always be modeled by recursion in functional programs.

Note also that the definition of `sqrtIter` contains a return type, which follows the parameter section. Such return types are mandatory for recursive functions. For a non-recursive function, the return type is optional; if it is missing the type checker will compute it from the type of the function's right-hand side. However, even for non-recursive functions it is often a good idea to include a return type for better documentation.

As a second step, we define the two functions called by `sqrtIter`: a function to `improve` the guess and a termination test `isGoodEnough`. Here is their definition.

```
def improve(guess: double, x: double) =
  (guess + x / guess) / 2

def isGoodEnough(guess: double, x: double) =
  abs(square(guess) - x) < 0.001
```

Finally, the `sqrt` function itself is defined by an application of `sqrtIter`.

```
def sqrt(x: double) = sqrtIter(1.0, x)
```

Exercise 3.4.1 The `isGoodEnough` test is not very precise for small numbers and might lead to non-termination for very large ones (why?). Design a different version of `isGoodEnough` which does not have these problems.

Exercise 3.4.2 Trace the execution of the `sqrt(4)` expression.

3.5 Nested Functions

The functional programming style encourages the construction of many small helper functions. In the last example, the implementation of `sqrt` made use of the helper functions `sqrtIter`, `improve` and `isGoodEnough`. The names of these functions are relevant only for the implementation of `sqrt`. We normally do not want users of `sqrt` to access these functions directly.

We can enforce this (and avoid name-space pollution) by including the helper functions within the calling function itself:

```
def sqrt(x: double) = {
  def sqrtIter(guess: double, x: double): double =
      if (isGoodEnough(guess, x)) guess
      else sqrtIter(improve(guess, x), x)
    def improve(guess: double, x: double) =
      (guess + x / guess) / 2
    def isGoodEnough(guess: double, x: double) =
      abs(square(guess) - x) < 0.001
    sqrtIter(1.0, x)
  }
```

In this program, the braces `{ ... }` enclose a *block*. Blocks in Scala are themselves expressions. Every block ends in a result expression which defines its value. The result expression may be preceded by auxiliary definitions, which are visible only in the block itself.

Every definition in a block must be followed by a semicolon, which separates this definition from subsequent definitions or the result expression. However, a semicolon is inserted implicitly at the end of each line, unless one of the following conditions is true.

1. Either the line in question ends in a word such as a period or an infix-operator which would not be legal as the end of an expression.
2. Or the next line begins with a word that cannot start a expression.
3. Or we are inside parentheses (...) or brackets , because these cannot contain multiple statements anyway.

Therefore, the following are all legal:

```
def f(x: int) = x + 1;
f(1) + f(2)

def g1(x: int) = x + 1
g(1) + g(2)

def g2(x: int) = {x + 1};  /* ‘;’ mandatory */ g2(1) + g2(2)

def h1(x) =
  x +
  y
h1(1) * h1(2)

def h2(x: int) = (
  x     // parentheses mandatory, otherwise a semicolon
  + y   // would be inserted after the ‘x’.
)
h2(1) / h2(2)
```

Scala uses the usual block-structured scoping rules. A name defined in some outer block is visible also in some inner block, provided it is not redefined there. This rule permits us to simplify our `sqrt` example. We need not pass x around as an additional parameter of the nested functions, since it is always visible in them as a parameter of the outer function `sqrt`. Here is the simplified code:

```
def sqrt(x: double) = {
  def sqrtIter(guess: double): double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess))
  def improve(guess: double) =
    (guess + x / guess) / 2
  def isGoodEnough(guess: double) =
    abs(square(guess) - x) < 0.001
  sqrtIter(1.0)
}
```

3.6 Tail Recursion

Consider the following function to compute the greatest common divisor of two given numbers.

```
def gcd(a: int, b: int): int = if (b == 0) a else gcd(b, a % b)
```

Using our substitution model of function evaluation, `gcd(14, 21)` evaluates as follows:

```
        gcd(14, 21)
→       if (21 == 0) 14 else gcd(21, 14 % 21)
→       if (false) 14 else gcd(21, 14 % 21)
→       gcd(21, 14 % 21)
→       gcd(21, 14)
→       if (14 == 0) 21 else gcd(14, 21 % 14)
→ →     gcd(14, 21 % 14)
→       gcd(14, 7)
→       if (7 == 0) 14 else gcd(7, 14 % 7)
→ →     gcd(7, 14 % 7)
→       gcd(7, 0)
→       if (0 == 0) 7 else gcd(0, 7 % 0)
→ →     7
```

Contrast this with the evaluation of another recursive function, `factorial`:

```
def factorial(n: int): int = if (n == 0) 1 else n * factorial(n - 1)
```

The application `factorial(5)` rewrites as follows:

```
          factorial(5)
→         if (5 == 0) 1 else 5 * factorial(5 - 1)
→         5 * factorial(5 - 1)
→         5 * factorial(4)
→ ... →   5 * (4 * factorial(3))
→ ... →   5 * (4 * (3 * factorial(2)))
→ ... →   5 * (4 * (3 * (2 * factorial(1))))
→ ... →   5 * (4 * (3 * (2 * (1 * factorial(0))))
→ ... →   5 * (4 * (3 * (2 * (1 * 1))))
→ ... →   120
```

There is an important difference between the two rewrite sequences: The terms in the rewrite sequence of `gcd` have again and again the same form. As evaluation proceeds, their size is bounded by a constant. By contrast, in the evaluation of factorial we get longer and longer chains of operands which are then multiplied in the last part of the evaluation sequence.

Even though actual implementations of Scala do not work by rewriting terms, they nevertheless should have the same space behavior as in the rewrite sequences. In the implementation of `gcd`, one notes that the recursive call to `gcd` is the last action performed in the evaluation of its body. One also says that `gcd` is "tail-recursive". The final call in a tail-recursive function can be implemented by a jump back to the beginning of that function. The arguments of that call can overwrite the parameters of the current instantiation of `gcd`, so that no new stack space is needed. Hence, tail recursive functions are iterative processes, which can be executed in constant space.

By contrast, the recursive call in `factorial` is followed by a multiplication. Hence, a new stack frame is allocated for the recursive instance of factorial, and is deallocated after that instance has finished. The given formulation of the factorial function is not tail-recursive; it needs space proportional to its input parameter for its execution.

More generally, if the last action of a function is a call to another (possibly the same) function, only a single stack frame is needed for both functions. Such calls are called "tail calls". In principle, tail calls can always re-use the stack frame of the calling function. However, some run-time environments (such as the Java VM) lack the primitives to make stack frame re-use for tail calls efficient. A production quality Scala implementation is therefore only required to re-use the stack frame of a directly tail-recursive function whose last action is a call to itself. Other tail calls might be optimized also, but one should not rely on this across implementations.

Exercise 3.6.1 Design a tail-recursive version of `factorial`.

Chapter 4

First-Class Functions

A function in Scala is a “first-class value”. Like any other value, it may be passed as a parameter or returned as a result. Functions which take other functions as parameters or return them as results are called *higher-order* functions. This chapter introduces higher-order functions and shows how they provide a flexible mechanism for program composition.

As a motivating example, consider the following three related tasks:

1. Write a function to sum all integers between two given numbers a and b:

   ```
   def sumInts(a: int, b: int): int =
     if (a > b) 0 else a + sumInts(a + 1, b)
   ```

2. Write a function to sum the squares of all integers between two given numbers a and b:

   ```
   def square(x: int): int = x * x
   def sumSquares(a: int, b: int): int =
     if (a > b) 0 else square(a) + sumSquares(a + 1, b)
   ```

3. Write a function to sum the powers 2^n of all integers n between two given numbers a and b:

   ```
   def powerOfTwo(x: int): int = if (x == 0) 1 else x * powerOfTwo(x - 1)
   def sumPowersOfTwo(a: int, b: int): int =
     if (a > b) 0 else powerOfTwo(a) + sumPowersOfTwo(a + 1, b)
   ```

These functions are all instances of $\sum_a^b f(n)$ for different values of f. We can factor out the common pattern by defining a function sum:

```
def sum(f: int => int, a: int, b: int): double =
  if (a > b) 0 else f(a) + sum(f, a + 1, b)
```

The type `int => int` is the type of functions that take arguments of type `int` and return results of type `int`. So `sum` is a function which takes another function as a parameter. In other words, `sum` is a *higher-order* function.

Using `sum`, we can formulate the three summing functions as follows.

```
def sumInts(a: int, b: int): int = sum(id, a, b)
def sumSquares(a: int, b: int): int = sum(square, a, b)
def sumPowersOfTwo(a: int, b: int): int = sum(powerOfTwo, a, b)
```

where

```
def id(x: int): int = x
def square(x: int): int = x * x
def powerOfTwo(x: int): int = if (x == 0) 1 else x * powerOfTwo(x - 1)
```

4.1 Anonymous Functions

Parameterization by functions tends to create many small functions. In the previous example, we defined `id`, `square` and `power` as separate functions, so that they could be passed as arguments to `sum`.

Instead of using named function definitions for these small argument functions, we can formulate them in a shorter way as *anonymous functions*. An anonymous function is an expression that evaluates to a function; the function is defined without giving it a name. As an example consider the anonymous square function:

```
(x: int) => x * x
```

The part before the arrow '=>' is the parameter of the function, whereas the part following the '=>' is its body. If there are several parameters, we need to enclose them in parentheses. For instance, here is an anonymous function which multiples its two arguments.

```
(x: int, y: int) => x * y
```

Using anonymous functions, we can reformulate the first two summation functions without named auxiliary functions:

```
def sumInts(a: int, b: int): int = sum(x: int => x, a, b)
def sumSquares(a: int, b: int): int = sum(x: int => x * x, a, b)
```

Often, the Scala compiler can deduce the parameter type(s) from the context of the anonymous function in which case they can be omitted. For instance, in the case of `sumInts` or `sumSquares`, one knows from the type of `sum` that the first parameter must be a function of type `int => int`. Hence, the parameter type `int` is redundant and may be omitted. If there is a single parameter without a type, we may also omit the parentheses around it:

```
def sumInts(a: int, b: int): int = sum(x => x, a, b)
def sumSquares(a: int, b: int): int = sum(x => x * x, a, b)
```

Generally, the Scala term `(x`$_1$`: T`$_1$`, ..., x`$_n$`: T`$_n$`) => E` defines a function which maps its parameters `x`$_1$`, ..., x`$_n$ to the result of the expression `E` (where `E` may refer to `x`$_1$`, ..., x`$_n$). Anonymous functions are not essential language elements of Scala, as they can always be expressed in terms of named functions. Indeed, the anonymous function

```
(x₁: T₁, ..., xₙ: Tₙ) => E
```

is equivalent to the block

```
{ def f (x₁: T₁, ..., xₙ: Tₙ) = E ; f }
```

where `f` is fresh name which is used nowhere else in the program. We also say, anonymous functions are "syntactic sugar".

4.2 Currying

The latest formulation of the summing functions is already quite compact. But we can do even better. Note that `a` and `b` appear as parameters and arguments of every function but they do not seem to take part in interesting combinations. Is there a way to get rid of them?

Let's try to rewrite `sum` so that it does not take the bounds `a` and `b` as parameters:

```
def sum(f: int => int): (int, int) => int = {
  def sumF(a: int, b: int): int =
    if (a > b) 0 else f(a) + sumF(a + 1, b)
  &sumF
}
```

In this formulation, `sum` is a function which returns another function, namely the specialized summing function `sumF`. This latter function does all the work; it takes the bounds `a` and `b` as parameters, applies `sum`'s function parameter `f` to all integers between them, and sums up the results.

Using this new formulation of `sum`, we can now define:

```
def sumInts  =  &sum(x => x)
def sumSquares  =  &sum(x => x * x)
def sumPowersOfTwo  =  &sum(powerOfTwo)
```

Or, equivalently, with value definitions:

```
val sumInts  =  &sum(x => x)
val sumSquares  =  &sum(x => x * x)
val sumPowersOfTwo  =  &sum(powerOfTwo)
```

Note the prefix operator `&` in front of the right-hand sides of the definitions above. This operator expresses that the partial applications of `sum` should be treated as function values. If it is omitted, the Scala compiler would complain that the applications of `sum` lack some of their arguments. The `&` operator can however be omitted if the expected type of an expression is a function type (for instance, this was the case in for `sumF` expression in the last example).

`sumInts`, `sumSquares`, and `sumPowersOfTwo` can be applied like any other function. For instance,

```
> sumSquares(1, 10) + sumPowersOfTwo(10, 20)
267632001: scala.Int
```

How are function-returning functions applied? As an example, in the expression

```
sum(x => x * x)(1, 10) ,
```

the function `sum` is applied to the squaring function `(x => x * x)`. The resulting function is then applied to the second argument list, `(1, 10)`.

This notation is possible because function application associates to the left. That is, if args_1 and args_2 are argument lists, then

$$f(\text{args}_1)(\text{args}_2) \quad \text{is equivalent to} \quad (f(\text{args}_1))(\text{args}_2)$$

In our example, `sum(x => x * x)(1, 10)` is equivalent to the following expression: `(sum(x => x * x))(1, 10)`.

The style of function-returning functions is so useful that Scala has special syntax for it. For instance, the next definition of `sum` is equivalent to the previous one, but is shorter:

```
def sum(f: int => int)(a: int, b: int): int =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)
```

Generally, a curried function definition

```
def f (args₁) ... (argsₙ) = E
```

where $n > 1$ expands to

```
def f (args₁) ... (argsₙ₋₁) = { def g (argsₙ) = E ; g }
```

where `g` is a fresh identifier. Or, shorter, using an anonymous function:

```
def f (args₁) ... (argsₙ₋₁) = ( argsₙ ) => E .
```

Performing this step n times yields that

```
def f (args₁) ... (argsₙ) = E
```

is equivalent to

```
def f = (args₁) => ... => (argsₙ) => E .
```

Or, equivalently, using a value definition:

```
val f = (args₁) => ... => (argsₙ) => E .
```

This style of function definition and application is called *currying* after its promoter, Haskell B. Curry, a logician of the 20th century, even though the idea goes back further to Moses Schönfinkel and Gottlob Frege.

The type of a function-returning function is expressed analogously to its parameter list. Taking the last formulation of `sum` as an example, the type of `sum` is `(int => int) => (int, int) => int`. This is possible because function types associate to the right. I.e.

```
T₁ => T₂ => T₃       is equivalent to     T₁ => (T₂ => T₃)
```

Exercise 4.2.1 1. The `sum` function uses a linear recursion. Can you write a tail-recursive one by filling in the ??'s?

```
def sum(f: int => double)(a: int, b: int): double = {
  def iter(a, result) = {
    if (??) ??
    else iter(??, ??)
  }
  iter(??, ??)
}
```

Exercise 4.2.2 Write a function `product` that computes the product of the values of functions at points over a given range.

Exercise 4.2.3 Write `factorial` in terms of `product`.

Exercise 4.2.4 Can you write an even more general function which generalizes both `sum` and `product`?

4.3 Example: Finding Fixed Points of Functions

A number `x` is called a *fixed point* of a function `f` if

```
f(x) = x .
```

For some functions `f` we can locate the fixed point by beginning with an initial guess and then applying `f` repeatedly, until the value does not change anymore (or the change is within a small tolerance). This is possible if the sequence

```
x, f(x), f(f(x)), f(f(f(x))), ...
```

converges to fixed point of f. This idea is captured in the following "fixed-point finding function":

```
val tolerance = 0.0001
def isCloseEnough(x: double, y: double) = abs((x - y) / x) < tolerance
def fixedPoint(f: double => double)(firstGuess: double) = {
  def iterate(guess: double): double = {
    val next = f(guess)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
  iterate(firstGuess)
}
```

We now apply this idea in a reformulation of the square root function. Let's start with a specification of `sqrt`:

```
sqrt(x)  =  the y such that  y * y = x
         =  the y such that  y = x / y
```

Hence, `sqrt(x)` is a fixed point of the function `y => x / y`. This suggests that `sqrt(x)` can be computed by fixed point iteration:

```
def sqrt(x: double) = fixedPoint(y => x / y)(1.0)
```

But if we try this, we find that the computation does not converge. Let's instrument the fixed point function with a print statement which keeps track of the current guess value:

```
def fixedPoint(f: double => double)(firstGuess: double) = {
  def iterate(guess: double): double = {
    val next = f(guess)
    System.out.println(next)
    if (isCloseEnough(guess, next)) next
    else iterate(next)
  }
```

```
    iterate(firstGuess)
  }
```

Then, `sqrt(2)` yields:

```
    2.0
    1.0
    2.0
    1.0
    2.0
    ...
```

One way to control such oscillations is to prevent the guess from changing too much. This can be achieved by *averaging* successive values of the original sequence:

```
  > def sqrt(x: double) = fixedPoint(y => (y + x/y) / 2)(1.0)
  def sqrt(x: scala.Double): scala.Double
  > sqrt(2.0)
    1.5
    1.4166666666666665
    1.4142156862745097
    1.4142135623746899
    1.4142135623746899
```

In fact, expanding the `fixedPoint` function yields exactly our previous definition of fixed point from Section 3.4.

The previous examples showed that the expressive power of a language is considerably enhanced if functions can be passed as arguments. The next example shows that functions which return functions can also be very useful.

Consider again fixed point iterations. We started with the observation that $\sqrt(x)$ is a fixed point of the function `y => x / y`. Then we made the iteration converge by averaging successive values. This technique of *average damping* is so general that it can be wrapped in another function.

```
  def averageDamp(f: double => double)(x: double) = (x + f(x)) / 2
```

Using `averageDamp`, we can reformulate the square root function as follows.

```
  def sqrt(x: double) = fixedPoint(averageDamp(y => x/y))(1.0)
```

This expresses the elements of the algorithm as clearly as possible.

Exercise 4.3.1 Write a function for cube roots using `fixedPoint` and `averageDamp`.

4.4 Summary

We have seen in the previous chapter that functions are essential abstractions, because they permit us to introduce general methods of computing as explicit, named elements in our programming language. The present chapter has shown that these abstractions can be combined by higher-order functions to create further abstractions. As programmers, we should look out for opportunities to abstract and to reuse. The highest possible level of abstraction is not always the best, but it is important to know abstraction techniques, so that one can use abstractions where appropriate.

4.5 Language Elements Seen So Far

Chapters 3 and 4 have covered Scala's language elements to express expressions and types comprising of primitive data and functions. The context-free syntax of these language elements is given below in extended Backus-Naur form, where '`|`' denotes alternatives, `[...]` denotes option (0 or 1 occurrence), and `{...}` denotes repetition (0 or more occurrences).

Characters

Scala programs are sequences of (Unicode) characters. We distinguish the following character sets:

- whitespace, such as '', tabulator, or newline characters,
- letters '`a`' to '`z`', '`A`' to '`Z`',
- digits '`0`' to '`9`',
- the delimiter characters

```
.   ,   ;   (   )   {   }   [   ]   \   "   '
```

- operator characters, such as '`#`' '`+`', '`:`'. Essentially, these are printable characters which are in none of the character sets above.

Lexemes:

```
ident    =  letter {letter | digit}
         |   operator { operator }
         |   ident '_' ident
literal  =  “as in Java”
```

Literals are as in Java. They define numbers, characters, strings, or boolean values. Examples of literals as `0`, `1.0d10`, `'x'`, `"he said "hi!""`, or **`true`**.

Identifiers can be of two forms. They either start with a letter, which is followed by a (possibly empty) sequence of letters or symbols, or they start with an operator character, which is followed by a (possibly empty) sequence of operator characters. Both forms of identifiers may contain underscore characters '_'. Furthermore, an underscore character may be followed by either sort of identifier. Hence, the following are all legal identifiers:

```
x      Room10a      +      --      foldl_:      +_vector
```

It follows from this rule that subsequent operator-identifiers need to be separated by whitespace. For instance, the input `x+-y` is parsed as the three token sequence `x`, `+-`, `y`. If we want to express the sum of `x` with the negated value of `y`, we need to add at least one space, e.g. `x+ -y`.

The `$` character is reserved for compiler-generated identifiers; it should not be used in source programs.

The following are reserved words, they may not be used as identifiers:

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

Types:

```
Type          =  SimpleType | FunctionType
FunctionType  =  SimpleType '=>' Type | '(' [Types] ')' '=>' Type
SimpleType    =  byte | short | char | int | long | double | float |
                 boolean | unit | String
Types         =  Type {',' Type}
```

Types can be:

- number types `byte`, `short`, `char`, `int`, `long`, `float` and `double` (these are as in Java),
- the type `boolean` with values **`true`** and **`false`**,
- the type `unit` with the only value `{}`,
- the type `String`,
- function types such as `(int, int) => int` or `String => int => String`.

Expressions:

```
Expr         = InfixExpr | FunctionExpr | if '(' Expr ')' Expr else Expr
InfixExpr    = PrefixExpr | InfixExpr Operator InfixExpr
Operator     = ident
PrefixExpr   = ['+' | '-' | '!' | '~' ] SimpleExpr
SimpleExpr   = ident | literal | SimpleExpr '.' ident | Block
FunctionExpr = Bindings '=>' Expr
Bindings     = ident [':' SimpleType] | '(' [Binding {',' Binding}] ')'
Binding      = ident [':' Type]
Block        = '{' {Def ';'} Expr '}'
```

Expressions can be:

- identifiers such as `x`, `isGoodEnough`, `*`, or `+-`,
- literals, such as `0`, `1.0`, or `"abc"`,
- field and method selections, such as `System.out.println`,
- function applications, such as `sqrt(x)`,
- operator applications, such as `-x` or `y + x`,
- conditionals, such as `if (x < 0) -x else x`,
- blocks, such as `{ val x = abs(y) ; x * 2 }`,
- anonymous functions, such as `x => x + 1` or `(x: int, y: int) => x + y`.

Definitions:

```
Def          =  FunDef  |  ValDef
FunDef       =  'def' ident {'(' [Parameters] ')'} [':' Type] '=' Expr
ValDef       =  'val' ident [':' Type] '=' Expr
Parameters   =  Parameter {',' Parameter}
Parameter    =  ident ':' ['=>'] Type
```

Definitions can be:

- function definitions such as `def square(x: int): int = x * x`,
- value definitions such as `val y = square(2)`.

Chapter 5

Classes and Objects

Scala does not have a built-in type of rational numbers, but it is easy to define one, using a class. Here's a possible implementation.

```
class Rational(n: int, d: int) {
  private def gcd(x: int, y: int): int = {
    if (x == 0) y
    else if (x < 0) gcd(-x, y)
    else if (y < 0) -gcd(x, -y)
    else gcd(y % x, x)
  }
  private val g = gcd(n, d)

  val numer: int = n/g
  val denom: int = d/g
  def +(that: Rational) =
    new Rational(numer * that.denom + that.numer * denom,
                 denom * that.denom)
  def -(that: Rational) =
    new Rational(numer * that.denom - that.numer * denom,
                 denom * that.denom)
  def *(that: Rational) =
    new Rational(numer * that.numer, denom * that.denom)
  def /(that: Rational) =
    new Rational(numer * that.denom, denom * that.numer)
}
```

This defines `Rational` as a class which takes two constructor arguments `n` and `d`, containing the number's numerator and denominator parts. The class provides fields which return these parts as well as methods for arithmetic over rational numbers. Each arithmetic method takes as parameter the right operand of the operation. The left operand of the operation is always the rational number of which the method is a member.

Private members. The implementation of rational numbers defines a private method `gcd` which computes the greatest common denominator of two integers, as well as a private field `g` which contains the `gcd` of the constructor arguments. These members are inaccessible outside class `Rational`. They are used in the implementation of the class to eliminate common factors in the constructor arguments in order to ensure that numerator and denominator are always in normalized form.

Creating and Accessing Objects. As an example of how rational numbers can be used, here's a program that prints the sum of all numbers $1/i$ where i ranges from 1 to 10.

```
var i = 1
var x = new Rational(0, 1)
while (i <= 10) {
  x = x + new Rational(1,i)
  i = i + 1
}
System.out.println("" + x.numer + "/" + x.denom)
```

The + takes as left operand a string and as right operand a value of arbitrary type. It returns the result of converting its right operand to a string and appending it to its left operand.

Inheritance and Overriding. Every class in Scala has a superclass which it extends. If a class does not mention a superclass in its definition, the root type `scala.AnyRef` is implicitly assumed (for Java implementations, this type is an alias for `java.lang.Object`. For instance, class `Rational` could equivalently be defined as

```
class Rational(n: int, d: int) extends AnyRef {
  ... // as before
}
```

A class inherits all members from its superclass. It may also redefine (or: *override*) some inherited members. For instance, class `java.lang.Object` defines a method `toString` which returns a representation of the object as a string:

```
class Object {
  ...
  def toString: String = ...
}
```

The implementation of `toString` in `Object` forms a string consisting of the object's class name and a number. It makes sense to redefine this method for objects that are rational numbers:

```
class Rational(n: int, d: int) extends AnyRef {
  ... // as before
  override def toString = "" + numer + "/" + denom
}
```

Note that, unlike in Java, redefining definitions need to be preceded by an **override** modifier.

If class A extends class B, then objects of type A may be used wherever objects of type B are expected. We say in this case that type A *conforms* to type B. For instance, `Rational` conforms to `AnyRef`, so it is legal to assign a `Rational` value to a variable of type `AnyRef`:

```
var x: AnyRef = new Rational(1,2)
```

Parameterless Methods. Unlike in Java, methods in Scala do not necessarily take a parameter list. An example is the `square` method below. This method is invoked by simply mentioning its name.

```
class Rational(n: int, d: int) extends AnyRef {
  ... // as before
  def square = new Rational(numer*numer, denom*denom)
}
val r = new Rational(3,4)
System.out.println(r.square);           // prints‘‘9/16’’*
```

That is, parameterless methods are accessed just as value fields such as `numer` are. The difference between values and parameterless methods lies in their definition. The right-hand side of a value is evaluated when the object is created, and the value does not change afterwards. A right-hand side of a parameterless method, on the other hand, is evaluated each time the method is called. The uniform access of fields and parameterless methods gives increased flexibility for the implementer of a class. Often, a field in one version of a class becomes a computed value in the next version. Uniform access ensures that clients do not have to be rewritten because of that change.

Abstract Classes. Consider the task of writing a class for sets of integer numbers with two operations, `incl` and `contains`. `(s incl x)` should return a new set which contains the element `x` together with all the elements of set `s`. `(s contains x)` should return true if the set `s` contains the element `x`, and should return **false** otherwise. The interface of such sets is given by:

```
abstract class IntSet {
  def incl(x: int): IntSet
  def contains(x: int): boolean
}
```

`IntSet` is labeled as an *abstract class*. This has two consequences. First, abstract classes may have *deferred* members which are declared but which do not have an implementation. In our case, both `incl` and `contains` are such members. Second, because an abstract class might have unimplemented members, no objects of that class may be created using **new**. By contrast, an abstract class may be used as a base class of some other class, which implements the deferred members.

Traits. Instead of **`abstract class`** one also often uses the keyword **`trait`** in Scala. Traits are abstract classes that are meant to be added to some other class. This might be because a trait adds some methods or fields to an unknown parent class. For instance, a trait `Bordered` might be used to add a border to a various graphical components. Another usage scenario is where the trait collects signatures of some functionality provided by different classes, much in the way a Java interface would work.

Since `IntSet` falls in this category, one can alternatively define it as a trait:

```
trait IntSet {
  def incl(x: int): IntSet
  def contains(x: int): boolean
}
```

Implementing Abstract Classes. Let's say, we plan to implement sets as binary trees. There are two possible forms of trees. A tree for the empty set, and a tree consisting of an integer and two subtrees. Here are their implementations.

```
class EmptySet extends IntSet {
  def contains(x: int): boolean = false
  def incl(x: int): IntSet = new NonEmptySet(x, new EmptySet, new EmptySet)
}

class NonEmptySet(elem:int, left:IntSet, right:IntSet) extends IntSet {
  def contains(x: int): boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: int): IntSet =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

Both `EmptySet` and `NonEmptySet` extend class `IntSet`. This implies that types `EmptySet` and `NonEmptySet` conform to type `IntSet` – a value of type `EmptySet` or `NonEmptySet` may be used wherever a value of type `IntSet` is required.

Exercise 5.0.1 Write methods `union` and `intersection` to form the union and intersection between two sets.

Exercise 5.0.2 Add a method

```
def excl(x: int)
```

to return the given set without the element x. To accomplish this, it is useful to also implement a test method

```
def isEmpty: boolean
```

for sets.

Dynamic Binding. Object-oriented languages (Scala included) use *dynamic dispatch* for method invocations. That is, the code invoked for a method call depends on the run-time type of the object which contains the method. For example, consider the expression `s contains 7` where `s` is a value of declared type `s: IntSet`. Which code for `contains` is executed depends on the type of value of `s` at run-time. If it is an `EmptySet` value, it is the implementation of `contains` in class `EmptySet` that is executed, and analogously for `NonEmptySet` values. This behavior is a direct consequence of our substitution model of evaluation. For instance,

```
    (new EmptySet).contains(7)

->                 (by replacing contains by its body in class EmptySet)

    false
```

Or,

```
    new NonEmptySet(7, new EmptySet, new EmptySet).contains(1)

->                 (by replacing contains by its body in class NonEmptySet)

    if (1 < 7) new EmptySet contains 1
    else if (1 > 7) new EmptySet contains 1
    else true

->                 (by rewriting the conditional)
```

```
      new EmptySet contains 1

->                (by replacing contains by its body in class EmptySet)

      false .
```

Dynamic method dispatch is analogous to higher-order function calls. In both cases, the identity of code to be executed is known only at run-time. This similarity is not just superficial. Indeed, Scala represents every function value as an object (see Section 7.6).

Objects. In the previous implementation of integer sets, empty sets were expressed with **new** `EmptySet`; so a new object was created every time an empty set value was required. We could have avoided unnecessary object creations by defining a value `empty` once and then using this value instead of every occurrence of **new** `EmptySet`. E.g.

```
val EmptySetVal = new EmptySet
```

One problem with this approach is that a value definition such as the one above is not a legal top-level definition in Scala; it has to be part of another class or object. Also, the definition of class `EmptySet` now seems a bit of an overkill – why define a class of objects, if we are only interested in a single object of this class? A more direct approach is to use an *object definition*. Here is a more streamlined alternative definition of the empty set:

```
object EmptySet extends IntSet {
  def contains(x: int): boolean = false
  def incl(x: int): IntSet = new NonEmptySet(x, EmptySet, EmptySet)
}
```

The syntax of an object definition follows the syntax of a class definition; it has an optional extends clause as well as an optional body. As is the case for classes, the extends clause defines inherited members of the object whereas the body defines overriding or new members. However, an object definition defines a single object only it is not possible to create other objects with the same structure using **new**. Therefore, object definitions also lack constructor parameters, which might be present in class definitions.

Object definitions can appear anywhere in a Scala program; including at top-level. Since there is no fixed execution order of top-level entities in Scala, one might ask exactly when the object defined by an object definition is created and initialized. The answer is that the object is created the first time one of its members is accessed. This strategy is called *lazy evaluation*.

Standard Classes. Scala is a pure object-oriented language. This means that every value in Scala can be regarded as an object. In fact, even primitive types such as `int` or `boolean` are not treated specially. They are defined as type aliases of Scala classes in module `Predef`:

```
type boolean = scala.Boolean
type int = scala.Int
type long = scala.Long
...
```

For efficiency, the compiler usually represents values of type `scala.Int` by 32 bit integers, values of type `scala.Boolean` by Java's booleans, etc. But it converts these specialized representations to objects when required, for instance when a primitive `int` value is passed to a function with a parameter of type `AnyRef`. Hence, the special representation of primitive values is just an optimization, it does not change the meaning of a program.

Here is a specification of class `Boolean`.

```
package scala
abstract class Boolean {
  def && (x: => Boolean): Boolean
  def || (x: => Boolean): Boolean
  def !                 : Boolean

  def == (x: Boolean)   : Boolean
  def != (x: Boolean)   : Boolean
  def <  (x: Boolean)   : Boolean
  def >  (x: Boolean)   : Boolean
  def <= (x: Boolean)   : Boolean
  def >= (x: Boolean)   : Boolean
}
```

Booleans can be defined using only classes and objects, without reference to a built-in type of booleans or numbers. A possible implementation of class `Boolean` is given below. This is not the actual implementation in the standard Scala library. For efficiency reasons the standard implementation uses built-in booleans.

```
package scala
abstract class Boolean {
  def ifThenElse(thenpart: => Boolean, elsepart: => Boolean)

  def && (x: => Boolean): Boolean  =  ifThenElse(x, false)
  def || (x: => Boolean): Boolean  =  ifThenElse(true, x)
  def !                 : Boolean  =  ifThenElse(false, true)

  def == (x: Boolean)   : Boolean  =  ifThenElse(x, x.!)
```

```
    def != (x: Boolean)   : Boolean  =  ifThenElse(x.!, x)
    def <  (x: Boolean)   : Boolean  =  ifThenElse(false, x)
    def >  (x: Boolean)   : Boolean  =  ifThenElse(x.!, false)
    def <= (x: Boolean)   : Boolean  =  ifThenElse(x, true)
    def >= (x: Boolean)   : Boolean  =  ifThenElse(true, x.!)
  }
  case object True extends Boolean {
    def ifThenElse(t: => Boolean, e: => Boolean) = t
  }
  case object False extends Boolean {
    def ifThenElse(t: => Boolean, e: => Boolean) = e
  }
```

Here is a partial specification of class `Int`.

```
  package scala
  abstract class Int extends AnyVal {
    def coerce: Long
    def coerce: Float
    def coerce: Double

    def + (that: Double): Double
    def + (that: Float): Float
    def + (that: Long): Long
    def + (that: Int): Int;       // analogous for -, *, /, %

    def << (cnt: Int): Int;       // analogous for >>, >>>

    def & (that: Long): Long
    def & (that: Int): Int;       // analogous for |, ^

    def == (that: Double): Boolean
    def == (that: Float): Boolean
    def == (that: Long): Boolean;  // analogous for !=, <, >, <=, >=
  }
```

Class `Int` can in principle also be implemented using just objects and classes, without reference to a built in type of integers. To see how, we consider a slightly simpler problem, namely how to implement a type `Nat` of natural (i.e. non-negative) numbers. Here is the definition of an abstract class `Nat`:

```
  abstract class Nat {
    def isZero: Boolean
    def predecessor: Nat
    def successor: Nat
    def + (that: Nat): Nat
    def - (that: Nat): Nat
```

```
}
```

To implement the operations of class Nat, we define a sub-object Zero and a subclass Succ (for successor). Each number N is represented as N applications of the Succ constructor to Zero:

$$\underbrace{\textit{new Succ(... new Succ}}_{N \text{ times}}\textit{(Zero) ...)}$$

The implementation of the Zero object is straightforward:

```
object Zero extends Nat {
  def isZero: Boolean = true
  def predecessor: Nat = throw new Error("negative number")
  def successor: Nat = new Succ(Zero)
  def + (that: Nat): Nat = that
  def - (that: Nat): Nat = if (that.isZero) Zero
                           else throw new Error("negative number")
}
```

The implementation of the predecessor and subtraction functions on Zero throws an Error exception, which aborts the program with the given error message.

Here is the implementation of the successor class:

```
class Succ(x: Nat) extends Nat  {
  def isZero: Boolean = false
  def predecessor: Nat = x
  def successor: Nat = new Succ(this)
  def + (that: Nat): Nat = x + that.successor
  def - (that: Nat): Nat = x - that.predecessor
}
```

Note the implementation of method successor. To create the successor of a number, we need to pass the object itself as an argument to the Succ constructor. The object itself is referenced by the reserved name **this**.

The implementations of + and - each contain a recursive call with the constructor argument as receiver. The recursion will terminate once the receiver is the Zero object (which is guaranteed to happen eventually because of the way numbers are formed).

Exercise 5.0.3 Write an implementation Integer of integer numbers The implementation should support all operations of class Nat while adding two methods

```
def isPositive: Boolean
def negate: Integer
```

The first method should return **true** if the number is positive. The second method should negate the number. Do not use any of Scala's standard numeric classes in your implementation. (Hint: There are two possible ways to implement `Integer`. One can either make use the existing implementation of `Nat`, representing an integer as a natural number and a sign. Or one can generalize the given implementation of `Nat` to `Integer`, using the three subclasses `Zero` for `0`, `Succ` for positive numbers and `Pred` for negative numbers.)

Language Elements Introduced In This Chapter

Types:

```
Type          = ...  |  ident
```

Types can now be arbitrary identifiers which represent classes.

Expressions:

```
Expr          = ...  |  Expr '.' ident  |  'new' Expr  |  'this'
```

An expression can now be an object creation, or a selection `E.m` of a member `m` from an object-valued expression `E`, or it can be the reserved name **this**.

Definitions and Declarations:

```
Def           = FunDef  |  ValDef  |  ClassDef  |  TraitDef  |  ObjectDef
ClassDef      = ['abstract'] 'class' ident ['(' [Parameters] ')']
                ['extends' Expr] ['{' {TemplateDef} '}']
TraitDef      = 'trait' ident ['extends' Expr] ['{' {TemplateDef} '}']
ObjectDef     = 'object' ident ['extends' Expr] ['{' {ObjectDef} '}']
TemplateDef   = [Modifier] (Def | Dcl)
ObjectDef     = [Modifier] Def
Modifier      = 'private'  |  'override'
Dcl           = FunDcl  |  ValDcl
FunDcl        = 'def' ident {'(' [Parameters] ')'} ':' Type
ValDcl        = 'val' ident ':' Type
```

A definition can now be a class, trait or object definition such as

```
class C(params) extends B { defs }
trait T extends B { defs }
object O extends B { defs }
```

The definitions `defs` in a class, trait or object may be preceded by modifiers **private** or **override**.

Abstract classes and traits may also contain declarations. These introduce *deferred* functions or values with their types, but do not give an implementation. Deferred members have to be implemented in subclasses before objects of an abstract class or trait can be created.

Chapter 6

Case Classes and Pattern Matching

Say, we want to write an interpreter for arithmetic expressions. To keep things simple initially, we restrict ourselves to just numbers and + operations. Such expressions can be represented as a class hierarchy, with an abstract base class `Expr` as the root, and two subclasses `Number` and `Sum`. Then, an expression `1 + (3 + 7)` would be represented as

```
new Sum(new Number(1), new Sum(new Number(3), new Number(7)))
```

Now, an evaluator of an expression like this needs to know of what form it is (either `Sum` or `Number`) and also needs to access the components of the expression. The following implementation provides all necessary methods.

```
abstract class Expr {
  def isNumber: boolean
  def isSum: boolean
  def numValue: int
  def leftOp: Expr
  def rightOp: Expr
}
class Number(n: int) extends Expr {
  def isNumber: boolean = true
  def isSum: boolean = false
  def numValue: int = n
  def leftOp: Expr = throw new Error("Number.leftOp")
  def rightOp: Expr = throw new Error("Number.rightOp")
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: boolean = false
  def isSum: boolean = true
```

```
    def numValue: int = throw new Error("Sum.numValue")
    def leftOp: Expr = e1
    def rightOp: Expr = e2
  }
```

With these classification and access methods, writing an evaluator function is simple:

```
  def eval(e: Expr): int = {
    if (e.isNumber) e.numValue
    else if (e.isSum) eval(e.leftOp) + eval(e.rightOp)
    else throw new Error("unrecognized expression kind")
  }
```

However, defining all these methods in classes `Sum` and `Number` is rather tedious. Furthermore, the problem becomes worse when we want to add new forms of expressions. For instance, consider adding a new expression form `Prod` for products. Not only do we have to implement a new class `Prod`, with all previous classification and access methods; we also have to introduce a new abstract method `isProduct` in class `Expr` and implement that method in subclasses `Number`, `Sum`, and `Prod`. Having to modify existing code when a system grows is always problematic, since it introduces versioning and maintenance problems.

The promise of object-oriented programming is that such modifications should be unnecessary, because they can be avoided by re-using existing, unmodified code through inheritance. Indeed, a more object-oriented decomposition of our problem solves the problem. The idea is to make the "high-level" operation `eval` a method of each expression class, instead of implementing it as a function outside the expression class hierarchy, as we have done before. Because `eval` is now a member of all expression nodes, all classification and access methods become superfluous, and the implementation is simplified considerably:

```
  abstract class Expr {
    def eval: int
  }
  class Number(n: int) extends Expr {
    def eval: int = n
  }
  class Sum(e1: Expr, e2: Expr) extends Expr {
    def eval: int = e1.eval + e2.eval
  }
```

Furthermore, adding a new `Prod` class does not entail any changes to existing code:

```
  class Prod(e1: Expr, e2: Expr) extends Expr {
    def eval: int = e1.eval * e2.eval
  }
```

The conclusion we can draw from this example is that object-oriented decomposition is the technique of choice for constructing systems that should be extensible with new types of data. But there is also another possible way we might want to extend the expression example. We might want to add new *operations* on expressions. For instance, we might want to add an operation that pretty-prints an expression tree to standard output.

If we have defined all classification and access methods, such an operation can easily be written as an external function. Here is an example:

```
def print(e: Expr) {
  if (e.isNumber) System.out.print(e.numValue)
  else if (e.isSum) {
    System.out.print("(");
    print(e.leftOp);
    System.out.print("+")
    print(e.rightOp)
    System.out.print(")")
  } else throw new Error("unrecognized expression kind")
}
```

However, if we had opted for an object-oriented decomposition of expressions, we would need to add a new `print` procedure to each class:

```
abstract class Expr {
  def eval: int
  def print
}
class Number(n: int) extends Expr {
  def eval: int = n
  def print { System.out.print(n) }
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: int = e1.eval + e2.eval
  def print {
    System.out.print("(");
    print(e1);
    System.out.print("+")
    print(e2)
    System.out.print(")")
  }
}
```

Hence, classical object-oriented decomposition requires modification of all existing classes when a system is extended with new operations.

As yet another way we might want to extend the interpreter, consider expression simplification. For instance, we might want to write a function which rewrites expressions of the form `a * b + a * c` to `a * (b + c)`. This operation requires inspection of more than a single node of the expression tree at the same time. Hence, it cannot be implemented by a method in each expression kind, unless that method can also inspect other nodes. So we are forced to have classification and access methods in this case. This seems to bring us back to square one, with all the problems of verbosity and extensibility.

Taking a closer look, one observers that the only purpose of the classification and access functions is to *reverse* the data construction process. They let us determine, first, which sub-class of an abstract base class was used and, second, what were the constructor arguments. Since this situation is quite common, Scala has a way to automate it with case classes.

6.1 Case Classes and Case Objects

Case classes and *case objects* are defined like a normal classes or objects, except that the definition is prefixed with the modifier **case**. For instance, the definitions

```
abstract class Expr
case class Number(n: int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

introduce `Number` and `Sum` as case classes. The **case** modifier in front of a class or object definition has the following effects.

1. Case classes implicitly come with a constructor function, with the same name as the class. In our example, the two functions

   ```
   def Number(n: int) = new Number(n)
   def Sum(e1: Expr, e2: Expr) = new Sum(e1, e2)
   ```

   would be added. Hence, one can now construct expression trees a bit more concisely, as in

   ```
   Sum(Sum(Number(1), Number(2)), Number(3))
   ```

2. Case classes and case objects implicitly come with implementations of methods `toString`, `equals` and `hashCode`, which override the methods with the same name in class `AnyRef`. The implementation of these methods takes in each case the structure of a member of a case class into account. The `toString` method represents an expression tree the way it was constructed. So,

   ```
   Sum(Sum(Number(1), Number(2)), Number(3))
   ```

would be converted to exactly that string, whereas the default implementation in class AnyRef would return a string consisting of the outermost constructor name Sum and a number. The equals methods treats two case members of a case class as equal if they have been constructed with the same constructor and with arguments which are themselves pairwise equal. This also affects the implementation of == and !=, which are implemented in terms of equals in Scala. So,

```
Sum(Number(1), Number(2)) == Sum(Number(1), Number(2))
```

will yield **true**. If Sum or Number were not case classes, the same expression would be **false**, since the standard implementation of equals in class AnyRef always treats objects created by different constructor calls as being different. The hashCode method follows the same principle as other two methods. It computes a hash code from the case class constructor name and the hash codes of the constructor arguments, instead of from the object's address, which is what the as the default implementation of hashCode does.

3. Case classes implicitly come with nullary accessor methods which retrieve the constructor arguments. In our example, Number would obtain an accessor method

   ```
   def n: int
   ```

   which returns the constructor parameter n, whereas Sum would obtain two accessor methods

   ```
   def e1: Expr, e2: Expr
   ```

   Hence, if for a value s of type Sum, say, one can now write s.e1, to access the left operand. However, for a value e of type Expr, the term e.e1 would be illegal since e1 is defined in Sum; it is not a member of the base class Expr. So, how do we determine the constructor and access constructor arguments for values whose static type is the base class Expr? This is solved by the fourth and final particularity of case classes.

4. Case classes allow the constructions of *patterns* which refer to the case class constructor.

6.2 Pattern Matching

Pattern matching is a generalization of C or Java's switch statement to class hierarchies. Instead of a switch statement, there is a standard method **match**, which is defined in Scala's root class Any, and therefore is available for all objects. The **match** method takes as argument a number of cases. For instance, here is an implementation of eval using pattern matching.

```
def eval(e: Expr): int = e match {
  case Number(x) => x
  case Sum(l, r) => eval(l) + eval(r)
}
```

In this example, there are two cases. Each case associates a pattern with an expression. Patterns are matched against the selector values `e`. The first pattern in our example, `Number(n)`, matches all values of the form `Number(v)`, where `v` is an arbitrary value. In that case, the *pattern variable* `n` is bound to the value `v`. Similarly, the pattern `Sum(l, r)` matches all selector values of form Sum(v_1, v_2) and binds the pattern variables `l` and `r` to v_1 and v_2, respectively.

In general, patterns are built from

- Case class constructors, e.g. `Number`, `Sum`, whose arguments are again patterns,
- pattern variables, e.g. `n`, `e1`, `e2`,
- the "wildcard" pattern _,
- literals, e.g. `1`, **true**, "abc",
- constant identifiers, e.g. `MAXINT`, `EmptySet`.

Pattern variables always start with a lower-case letter, so that they can be distinguished from constant identifiers, which start with an upper case letter. Each variable name may occur only once in a pattern. For instance, `Sum(x, x)` would be illegal as a pattern, since the pattern variable `x` occurs twice in it.

Meaning of Pattern Matching. A pattern matching expression

e **match** { **case** p_1 => e_1 ... **case** p_n => e_n }

matches the patterns $p_1, \ldots, p_n$ in the order they are written against the selector value `e`.

- A constructor pattern $C(p_1, \ldots, p_n)$ matches all values that are of type `C` (or a subtype thereof) and that have been constructed with `C`-arguments matching patterns $p_1, \ldots, p_n$.
- A variable pattern `x` matches any value and binds the variable name to that value.
- The wildcard pattern '_' matches any value but does not bind a name to that value.
- A constant pattern `C` matches a value which is equal (in terms of ==) to `C`.

The pattern matching expression rewrites to the right-hand-side of the first case whose pattern matches the selector value. References to pattern variables are replaced by corresponding constructor arguments. If none of the patterns matches, the pattern matching expression is aborted with a `MatchError` exception.

Example 6.2.1 Our substitution model of program evaluation extends quite naturally to pattern matching, For instance, here is how `eval` applied to a simple expression is re-written:

```
       eval(Sum(Number(1), Number(2)))

  ->                  (by rewriting the application)

       Sum(Number(1), Number(2)) match {
           case Number(n) => n
           case Sum(e1, e2) => eval(e1) + eval(e2)
       }

  ->                  (by rewriting the pattern match)

       eval(Number(1)) + eval(Number(2))

  ->                  (by rewriting the first application)

       Number(1) match {
           case Number(n) => n
           case Sum(e1, e2) => eval(e1) + eval(e2)
       } + eval(Number(2))

  ->                  (by rewriting the pattern match)

       1 + eval(Number(2))

  ->* 1 + 2 -> 3
```

Pattern Matching and Methods. In the previous example, we have used pattern matching in a function which was defined outside the class hierarchy over which it matches. Of course, it is also possible to define a pattern matching function in that class hierarchy itself. For instance, we could have defined `eval` is a method of the base class `Expr`, and still have used pattern matching in its implementation:

```
abstract class Expr {
  def eval: int = this match {
    case Number(n) => n
    case Sum(e1, e2) => e1.eval + e2.eval
    }
  }
```

Exercise 6.2.2 Consider the following definitions representing trees of integers. These definitions can be seen as an alternative representation of `IntSet`:

```
abstract class IntTree
case object EmptyTree extends IntTree
case class  Node(elem: int, left: IntTree, right: IntTree) extends IntTree
```

Complete the following implementations of function `contains` and `insert` for `IntTree`'s.

```
def contains(t: IntTree, v: int): boolean = t match { ...
  ...
}
def insert(t: IntTree, v: int): IntTree = t match { ...
  ...
}
```

Pattern Matching Anonymous Functions. So far, case-expressions always appeared in conjunction with a `match` operation. But it is also possible to use case-expressions by themselves. A block of case-expressions such as

{ **case** P_1 => E_1 ... **case** P_n => E_n }

is seen by itself as a function which matches its arguments against the patterns $P_1, \ldots, P_n$, and produces the result of one of $E_1, \ldots, E_n$. (If no pattern matches, the function would throw a `MatchError` exception instead). In other words, the expression above is seen as a shorthand for the anonymous function

(x => x **match** { **case** P_1 => E_1 ... **case** P_n => E_n })

where `x` is a fresh variable which is not used otherwise in the expression.

Chapter 7

Generic Types and Methods

Classes in Scala can have type parameters. We demonstrate the use of type parameters with functional stacks as an example. Say, we want to write a data type of stacks of integers, with methods `push`, `top`, `pop`, and `isEmpty`. This is achieved by the following class hierarchy:

```
abstract class IntStack {
  def push(x: int): IntStack = new IntNonEmptyStack(x, this)
  def isEmpty: boolean
  def top: int
  def pop: IntStack
}
class IntEmptyStack extends IntStack {
  def isEmpty = true
  def top = throw new Error("EmptyStack.top")
  def pop = throw new Error("EmptyStack.pop")
}
class IntNonEmptyStack(elem: int, rest: IntStack) {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

Of course, it would also make sense to define an abstraction for a stack of Strings. To do that, one could take the existing abstraction for `IntStack`, rename it to `StringStack` and at the same time rename all occurrences of type `int` to `String`.

A better way, which does not entail code duplication, is to parameterize the stack definitions with the element type. Parameterization lets us generalize from a specific instance of a problem to a more general one. So far, we have used parameterization only for values, but it is available also for types. To arrive at a *generic* version of `Stack`, we equip it with a type parameter.

```
abstract class Stack[a] {
  def push(x: a): Stack[a] = new NonEmptyStack[a](x, this)
  def isEmpty: boolean
  def top: a
  def pop: Stack[a]
}
class EmptyStack[a] extends Stack[a] {
  def isEmpty = true
  def top = throw new Error("EmptyStack.top")
  def pop = throw new Error("EmptyStack.pop")
}
class NonEmptyStack[a](elem: a, rest: Stack[a]) extends Stack[a] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

In the definitions above, 'a' is a *type parameter* of class `Stack` and its subclasses. Type parameters are arbitrary names; they are enclosed in brackets instead of parentheses, so that they can be easily distinguished from value parameters. Here is an example how the generic classes are used:

```
val x = new EmptyStack[int]
val y = x.push(1).push(2)
System.out.println(y.pop.top)
```

The first line creates a new empty stack of `int`'s. Note the actual type argument `[int]` which replaces the formal type parameter a.

It is also possible to parameterize methods with types. As an example, here is a generic method which determines whether one stack is a prefix of another.

```
def isPrefix[a](p: Stack[a], s: Stack[a]): boolean = {
  p.isEmpty ||
  p.top == s.top && isPrefix[a](p.pop, s.pop)
}
```

parameters are called *polymorphic*. Generic methods are also called *polymorphic*. The term comes from the Greek, where it means "having many forms". To apply a polymorphic method such as `isPrefix`, we pass type parameters as well as value parameters to it. For instance,

```
val s1 = new EmptyStack[String].push("abc")
val s2 = new EmptyStack[String].push("abx").push(s.pop)
System.out.println(isPrefix[String](s1, s2))
```

Local Type Inference. Passing type parameters such as [int] or [String] all the time can become tedious in applications where generic functions are used a lot. Quite often, the information in a type parameter is redundant, because the correct parameter type can also be determined by inspecting the function's value parameters or expected result type. Taking the expression isPrefix[String](s1, s2) as an example, we know that its value parameters are both of type Stack[String], so we can deduce that the type parameter must be String. Scala has a fairly powerful type inferencer which allows one to omit type parameters to polymorphic functions and constructors in situations like these. In the example above, one could have written isPrefix(s1, s2) and the missing type argument [String] would have been inserted by the type inferencer.

7.1 Type Parameter Bounds

Now that we know how to make classes generic it is natural to generalize some of the earlier classes we have written. For instance class IntSet could be generalized to sets with arbitrary element types. Let's try. The abstract class for generic sets is easily written.

```
abstract class Set[a] {
  def incl(x: a): Set[a];
  def contains(x: a): boolean;
}
```

However, if we still want to implement sets as binary search trees, we encounter a problem. The contains and incl methods both compare elements using methods < and >. For IntSet this was OK, since type int has these two methods. But for an arbitrary type parameter a, we cannot guarantee this. Therefore, the previous implementation of, say, contains would generate a compiler error.

```
  def contains(x: int): boolean =
    if (x < elem) left contains x
          ^ < not a member of type a.
```

One way to solve the problem is to restrict the legal types that can be substituted for type a to only those types that contain methods < and > of the correct types. There is a trait Ordered[a] in the standard class library Scala which represents values which are comparable (via < and >) to values of type a. This trait is defined as follows:

```
/** A class for totally ordered data. */
trait Ordered[a] {

  /** Result of comparing ‘this’ with operand ‘that’.
   *  returns ‘x’ where
   *  x < 0    iff    this < that
   *  x == 0   iff    this == that
   *  x > 0    iff    this > that
   */
  def compare(that: a): int;

  def <  (that: a): boolean = (this compare that) <  0
  def >  (that: a): boolean = (this compare that) >  0
  def <= (that: a): boolean = (this compare that) <= 0
  def >= (that: a): boolean = (this compare that) >= 0
  def compareTo(that: a): int = compare(that)
}
```

We can enforce the comparability of a type by demanding that the type is a subtype of `Ordered`. This is done by giving an upper bound to the type parameter of `Set`:

```
trait Set[a <: Ordered[a]] {
  def incl(x: a): Set[a];
  def contains(x: a): boolean;
}
```

The parameter declaration `a <: Ordered[a]` introduces `a` as a type parameter which must be a subtype of `Ordered[a]`, i.e. its values must be comparable to values of the same type.

With this restriction, we can now implement the rest of the generic set abstraction as we did in the case of `IntSets` before.

```
class EmptySet[a <: Ordered[a]] extends Set[a] {
  def contains(x: a): boolean = false
  def incl(x: a): Set[a] = new NonEmptySet(x, new EmptySet[a], new EmptySet[a])
}

class NonEmptySet[a <: Ordered[a]]
        (elem:a, left: Set[a], right: Set[a]) extends Set[a] {
  def contains(x: a): boolean =
    if (x < elem) left contains x
    else if (x > elem) right contains x
    else true
  def incl(x: a): Set[a] =
    if (x < elem) new NonEmptySet(elem, left incl x, right)
    else if (x > elem) new NonEmptySet(elem, left, right incl x)
    else this
}
```

Note that we have left out the type argument in the object creations **new** `NonEmptySet(...)`. In the same way as for polymorphic methods, missing type arguments in constructor calls are inferred from value arguments and/or the expected result type.

Here is an example that uses the generic set abstraction. Let's first create a subclass of `Ordered`, like this:

```
case class Num(value: double) extends Ordered[Num] {
  def compare(that: Num): int =
    if (this.value < that.value) -1
    else if (this.value > that.value) 1
    else 0
}
```

Then:

```
val s = new EmptySet[double].incl(Num(1.0)).incl(Num(2.0))
s.contains(1.5)
```

This is OK, as type `Num` implements the trait `Ordered[Num]`. However, the following example is in error.

```
val s = new EmptySet[java.io.File]
                      ^ java.io.File does not conform to type
                        parameter bound Ordered[java.io.File].
```

One probem with type parameter bounds is that they require forethought: if we had not declared `Num` a subclass of `Ordered`, we would not have been able to use `Num` elements in sets. By the same token, types inherited from Java, such as `int`, `double`, or `String` are not subclasses of `Ordered`, so values of these types cannot be used as set elements.

A more flexible design, which admits elements of these types, uses *view bounds* instead of the plain type bounds we have seen so far. The only change this entails in the example above is in the type parameters:

```
trait Set[a <% Ordered[a]] ...
class EmptySet[a <% Ordered[a]] ...
class NonEmptySet[a <% Ordered[a]] ...
```

View bounds `<%` are weaker than plain bounds `<:`: A view bounded type parameter clause `[a <% T]` only specifies that the bounded type a must be *convertible* to the bound type `T`, using an implicit conversion.

The Scala library predefines implicit conversions for a number of types, including the primitive types and `String`. Therefore, the redesign set abstraction can be instantiated with these types as well. More explanations on implicit conversions and view bounds are given in Section 23.

7.2 Variance Annotations

The combination of type parameters and subtyping poses some interesting questions. For instance, should Stack[String] be a subtype of Stack[AnyRef]? Intuitively, this seems OK, since a stack of Strings is a special case of a stack of AnyRefs. More generally, if T is a subtype of type S then Stack[T] should be a subtype of Stack[S]. This property is called *co-variant* subtyping.

In Scala, generic types have by default non-variant subtyping. That is, with Stack defined as above, stacks with different element types would never be in a subtype relation. However, we can enforce co-variant subtyping of stacks by changing the first line of the definition of class Stack as follows.

```
class Stack[+a] {
```

Prefixing a formal type parameter with a + indicates that subtyping is covariant in that parameter. Besides +, there is also a prefix - which indicates contra-variant subtyping. If Stack was defined **class** Stack[-a] ..., then T a subtype of type S would imply that Stack[S] is a subtype of Stack[T] (which in the case of stacks would be rather surprising!).

In a purely functional world, all types could be co-variant. However, the situation changes once we introduce mutable data. Consider the case of arrays in Java or .NET. Such arrays are represented in Scala by a generic class Array. Here is a partial definition of this class.

```
class Array[a] {
  def apply(index: int): a
  def update(index: int, elem: a)
}
```

The class above defines the way Scala arrays are seen from Scala user programs. The Scala compiler will map this abstraction to the underlying arrays of the host system in most cases where this possible.

In Java, arrays are indeed covariant; that is, for reference types T and S, if T is a subtype of S, then also Array[T] is a subtype of Array[S]. This might seem natural but leads to safety problems that require special runtime checks. Here is an example:

```
val x = new Array[String](1)
val y: Array[Any] = x
y(0) = new Rational(1, 2); // this is syntactic sugar for
                           // y.update(0, new Rational(1, 2))
```

In the first line, a new array of strings is created. In the second line, this array is bound to a variable y, of type Array[Any]. Assuming arrays are covariant, this is OK, since Array[String] is a subtype of Array[Any]. Finally, in the last line a rational number is stored in the array. This is also OK, since type Rational is a subtype of the element type `Any` of the array `y`. We thus end up storing a rational number in an array of strings, which clearly violates type soundness.

Java solves this problem by introducing a run-time check in the third line which tests whether the stored element is compatible with the element type with which the array was created. We have seen in the example that this element type is not necessarily the static element type of the array being updated. If the test fails, an `ArrayStoreException` is raised.

Scala solves this problem instead statically, by disallowing the second line at compile-time, because arrays in Scala have non-variant subtyping. This raises the question how a Scala compiler verifies that variance annotations are correct. If we had simply declared arrays co-variant, how would the potential problem have been detected?

Scala uses a conservative approximation to verify soundness of variance annotations. A covariant type parameter of a class may only appear in co-variant positions inside the class. Among the co-variant positions are the types of values in the class, the result types of methods in the class, and type arguments to other covariant types. Not co-variant are types of formal method parameters. Hence, the following class definition would have been rejected

```
class Array[+a] {
  def apply(index: int): a
  def update(index: int, elem: a)
                                ^ covariant type parameter a
                                  appears in contravariant position.
}
```

So far, so good. Intuitively, the compiler was correct in rejecting the `update` procedure in a co-variant class because `update` potentially changes state, and therefore undermines the soundness of co-variant subtyping.

However, there are also methods which do not mutate state, but where a type parameter still appears contra-variantly. An example is `push` in type `Stack`. Again the Scala compiler will reject the definition of this method for co-variant stacks.

```
class Stack[+a] {
  def push(x: a): Stack[a] =
              ^ covariant type parameter a
                appears in contravariant position.
```

This is a pity, because, unlike arrays, stacks are purely functional data structures and therefore should enable co-variant subtyping. However, there is a a way to solve the problem by using a polymorphic method with a lower type parameter bound.

7.3 Lower Bounds

We have seen upper bounds for type parameters. In a type parameter declaration such as `t <: U`, the type parameter `t` is restricted to range only over subtypes of type `U`. Symmetrical to this are lower bounds in Scala. In a type parameter declaration `t >: L`, the type parameter `t` is restricted to range only over *supertypes* of type `L`. (One can also combine lower and upper bounds, as in `t >: L <: U`.)

Using lower bounds, we can generalize the `push` method in `Stack` as follows.

```
class Stack[+a] {
  def push[b >: a](x: b): Stack[b] = new NonEmptyStack(x, this)
```

Technically, this solves our variance problem since now the type parameter `a` appears no longer as a parameter type of method `push`. Instead, it appears as lower bound for another type parameter of a method, which is classified as a co-variant position. Hence, the Scala compiler accepts the new definition of `push`.

In fact, we have not only solved the technical variance problem but also have generalized the definition of `push`. Before, we were required to push only elements with types that conform to the declared element type of the stack. Now, we can push also elements of a supertype of this type, but the type of the returned stack will change accordingly. For instance, we can now push an `AnyRef` onto a stack of `String`s, but the resulting stack will be a stack of `AnyRef`s instead of a stack of `String`s!

In summary, one should not hesitate to add variance annotations to your data structures, as this yields rich natural subtyping relationships. The compiler will detect potential soundness problems. Even if the compiler's approximation is too conservative, as in the case of method `push` of class `Stack`, this will often suggest a useful generalization of the contested method.

7.4 Least Types

Scala does not allow one to parameterize objects with types. That's why we originally defined a generic class `EmptyStack[a]`, even though a single value denoting empty stacks of arbitrary type would do. For co-variant stacks, however, one can use the following idiom:

```
object EmptyStack extends Stack[Nothing] { ... }
```

The bottom type `Nothing` contains no value, so the type `Stack[Nothing]` expresses the fact that an `EmptyStack` contains no elements. Furthermore, `Nothing` is a subtype of all other types. Hence, for co-variant stacks, `Stack[Nothing]` is a subtype of `Stack[T]`, for any other type `T`. This makes it possible to use a single empty stack object in user code. For instance:

```
val s = EmptyStack.push("abc").push(new AnyRef())
```

Let's analyze the type assignment for this expression in detail. The `EmptyStack` object is of type `Stack[Nothing]`, which has a method

```
push[b >: Nothing](elem: b): Stack[b] .
```

Local type inference will determine that the type parameter `b` should be instantiated to `String` in the application `EmptyStack.push("abc")`. The result type of that application is hence `Stack[String]`, which in turn has a method

```
push[b >: String](elem: b): Stack[b] .
```

The final part of the value definition above is the application of this method to **new** `AnyRef()`. Local type inference will determine that the type parameter `b` should this time be instantiated to `AnyRef`, with result type `Stack[AnyRef]`. Hence, the type assigned to value `s` is `Stack[AnyRef]`.

Besides `Nothing`, which is a subtype of every other type, there is also the type `Null`, which is a subtype of `scala.AnyRef`, and every class derived from it. The **`null`** literal in Scala is the only value of that type. This makes **`null`** compatible with every reference type, but not with a value type such as `int`.

We conclude this section with the complete improved definition of stacks. Stacks have now co-variant subtyping, the `push` method has been generalized, and the empty stack is represented by a single object.

```
abstract class Stack[+a] {
  def push[b >: a](x: b): Stack[b] = new NonEmptyStack(x, this)
  def isEmpty: boolean
  def top: a
  def pop: Stack[a]
}
object EmptyStack extends Stack[Nothing] {
  def isEmpty = true
  def top = throw new Error("EmptyStack.top")
  def pop = throw new Error("EmptyStack.pop")
}
class NonEmptyStack[+a](elem: a, rest: Stack[a]) extends Stack[a] {
  def isEmpty = false
  def top = elem
  def pop = rest
}
```

Many classes in the Scala library are generic. We now present two commonly used families of generic classes, tuples and functions. The discussion of another common class, lists, is deferred to the next chapter.

7.5 Tuples

Sometimes, a function needs to return more than one result. For instance, take the function `divmod` which returns the integer quotient and rest of two given integer arguments. Of course, one can define a class to hold the two results of `divmod`, as in:

```
case class TwoInts(first: int, second: int)
def divmod(x: int, y: int): TwoInts = new TwoInts(x / y, x % y)
```

However, having to define a new class for every possible pair of result types is very tedious. In Scala one can use instead a the generic classes `Tuple2`, which is defined as follows:

```
package scala
case class Tuple2[a, b](_1: a, _2: b)
```

With `Tuple2`, the `divmod` method can be written as follows.

```
def divmod(x: int, y: int) = new Tuple2[int, int](x / y, x % y)
```

As usual, type parameters to constructors can be omitted if they are deducible from value arguments. There exist also tuple classes for every other number of elements (the current Scala implementation limits this to tuples of some reasonable number of elements).

How are elements of tuples accessed? Since tuples are case classes, there are two possibilities. One can either access a tuple's fields using the names of the constructor parameters _i, as in the following example:

```
val xy = divmod(x, y)
System.out.println("quotient: " + x._1 + ", rest: " + x._2)
```

Or one uses pattern matching on tuples, as in the following example:

```
divmod(x, y) match {
  case Tuple2(n, d) =>
    System.out.println("quotient: " + n + ", rest: " + d)
}
```

Note that type parameters are never used in patterns; it would have been illegal to write case `Tuple2[int, int](n, d)`.

Tuples are so convenient that Scala defines special syntax for them. To form a tuple with n elements $x_1, \ldots, x_n$ one can write $(x_1, \ldots, x_n)$. This is equivalent to `Tuple`$n(x_1, \ldots, x_n)$. The (...) syntax works equivalently for types and for patterns. With that tuple syntax, the `divmod` example is written as follows:

```
def divmod(x: int, y: int): {int, int} = (x / y, x % y)
```

```
divmod(x, y) match {
  case (n, d) => System.out.println("quotient: " + n + ", rest: " + d)
}
```

7.6 Functions

Scala is a functional language in that functions are first-class values. Scala is also an object-oriented language in that every value is an object. It follows that functions are objects in Scala. For instance, a function from type `String` to type `int` is represented as an instance of the trait `Function1[String, int]`. The `Function1` trait is defined as follows.

```
package scala
trait Function1[-a, +b] {
  def apply(x: a): b
}
```

Besides `Function1`, there are also definitions of for functions of all other arities (the current implementation implements this only up to a reasonable limit). That is, there is one definition for each possible number of function parameters. Scala's function type syntax $T_1, \ldots, T_n$ `=>` S is simply an abbreviation for the parameterized type `Function`n`[`$T_1, \ldots, T_n, S$`]`.

Scala uses the same syntax $f(x)$ for function application, no matter whether f is a method or a function object. This is made possible by the following convention: A function application $f(x)$ where f is an object (as opposed to a method) is taken to be a shorthand for f`.apply(`x`)`. Hence, the `apply` method of a function type is inserted automatically where this is necessary.

That's also why we defined array subscripting in Section 7.2 by an `apply` method. For any array `a`, the subscript operation `a(i)` is taken to be a shorthand for `a.apply(i)`.

Functions are an example where a contra-variant type parameter declaration is useful. For example, consider the following code:

```
val f: (AnyRef => int)  =  x => x.hashCode()
val g: (String => int)  =  f
g("abc")
```

It's sound to bind the value `g` of type `String => int` to `f`, which is of type `AnyRef => int`. Indeed, all one can do with function of type `String => int` is pass it a string in order to obtain an integer. Clearly, the same works for function `f`: If we pass it a string (or any other object), we obtain an integer. This demonstrates that function subtyping is contra-variant in its argument type whereas it is covariant in its result type. In short, $S \Rightarrow T$ is a subtype of $S' \Rightarrow T'$, provided S' is a subtype of S

and T is a subtype of T'.

Example 7.6.1 Consider the Scala code

```
val plus1: (int => int)  =  (x: int) => x + 1
plus1(2)
```

This is expanded into the following object code.

```
val plus1: Function1[int, int] = new Function1[int, int] {
  def apply(x: int): int = x + 1
}
plus1.apply(2)
```

Here, the object creation `new Function1[int, int]{ ... }` represents an instance of an *anonymous class*. It combines the creation of a new `Function1` object with an implementation of the `apply` method (which is abstract in `Function1`). Equivalently, but more verbosely, one could have used a local class:

```
val plus1: Function1[int, int] = {
  class Local extends Function1[int, int] {
    def apply(x: int): int = x + 1
  }
  new Local: Function1[int, int]
}
plus1.apply(2)
```

Chapter 8

Lists

Lists are an important data structure in many Scala programs. A list containing the elements $x_1, \ldots, x_n$ is written `List(x₁, ..., xₙ)`. Examples are:

```
val fruit = List("apples", "oranges", "pears")
val nums  = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

Lists are similar to arrays in languages such as C or Java, but there are also three important differences. First, lists are immutable. That is, elements of a list cannot be changed by assignment. Second, lists have a recursive structure, whereas arrays are flat. Third, lists support a much richer set of operations than arrays usually do.

8.1 Using Lists

The List type. Like arrays, lists are *homogeneous*. That is, the elements of a list all have the same type. The type of a list with elements of type T is written List[T] (compare to T[] in Java).

```
val fruit: List[String]    = List("apples", "oranges", "pears")
val nums : List[int]       = List(1, 2, 3, 4)
val diag3: List[List[int]] = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty: List[int]       = List()
```

List constructors. All lists are built from two more fundamental constructors, Nil and :: (pronounced “cons”). Nil represents an empty list. The infix operator :: expresses list extension. That is, x :: xs represents a list whose first element is x, which is followed by (the elements of) list xs. Hence, the list values above could also have been defined as follows (in fact their previous definition is simply syntactic sugar for the definitions below).

```
val fruit  = "apples" :: ("oranges" :: ("pears" :: Nil))
val nums   = 1 :: (2 :: (3 :: (4 :: Nil)))
val diag3  = (1 :: (0 :: (0 :: Nil))) ::
             (0 :: (1 :: (0 :: Nil))) ::
             (0 :: (0 :: (1 :: Nil))) :: Nil
val empty  = Nil
```

The '`::`' operation associates to the right: `A :: B :: C` is interpreted as `A :: (B :: C)`. Therefore, we can drop the parentheses in the definitions above. For instance, we can write shorter

```
val nums  =  1 :: 2 :: 3 :: 4 :: Nil
```

Basic operations on lists. All operations on lists can be expressed in terms of the following three:

`head`	returns the first element of a list,
`tail`	returns the list consisting of all elements except the first element,
`isEmpty`	returns **true** iff the list is empty

These operations are defined as methods of list objects. So we invoke them by selecting from the list that's operated on. Examples:

```
empty.isEmpty   = true
fruit.isEmpty   = false
fruit.head      = "apples"
fruit.tail.head = "oranges"
diag3.head      = List(1, 0, 0)
```

The `head` and `tail` methods are defined only for non-empty lists. When selected from an empty list, they throw an exception.

As an example of how lists can be processed, consider sorting the elements of a list of numbers into ascending order. One simple way to do so is *insertion sort*, which works as follows: To sort a non-empty list with first element x and rest xs, sort the remainder xs and insert the element x at the right position in the result. Sorting an empty list will yield the empty list. Expressed as Scala code:

```
def isort(xs: List[int]): List[int] =
  if (xs.isEmpty) Nil
  else insert(xs.head, isort(xs.tail))
```

Exercise 8.1.1 Provide an implementation of the missing function `insert`.

List patterns. In fact, `::` is defined as a case class in Scala's standard library. Hence, it is possible to decompose lists by pattern matching, using patterns composed from the `Nil` and `::` constructors. For instance, `isort` can be written alternatively as follows.

```
def isort(xs: List[int]): List[int] = xs match {
  case List() => List()
  case x :: xs1 => insert(x, isort(xs1))
}
```

where

```
def insert(x: int, xs: List[int]): List[int] = xs match {
  case List() => List(x)
  case y :: ys => if (x <= y) x :: xs else y :: insert(x, ys)
}
```

8.2 Definition of class List I: First Order Methods

Lists are not built in in Scala; they are defined by an abstract class `List`, which comes with two subclasses for `::` and `Nil`. In the following we present a tour through class `List`.

```
package scala
abstract class List[+a] {
```

`List` is an abstract class, so one cannot define elements by calling the empty `List` constructor (e.g. by **new** `List`). The class has a type parameter a. It is co-variant in this parameter, which means that `List[S] <: List[T]` for all types `S` and `T` such that `S <: T`. The class is situated in the package `scala`. This is a package containing the most important standard classes of Scala. `List` defines a number of methods, which are explained in the following.

Decomposing lists. First, there are the three basic methods `isEmpty`, `head`, `tail`. Their implementation in terms of pattern matching is straightforward:

```
def isEmpty: boolean = this match {
  case Nil => true
  case x :: xs => false
}
def head: a = this match {
  case Nil => throw new Error("Nil.head")
  case x :: xs => x
}
def tail: List[a] = this match {
    case Nil => throw new Error("Nil.tail")
    case x :: xs => xs
  }
```

The next function computes the length of a list.

```
  def length = this match {
    case Nil => 0
    case x :: xs => 1 + xs.length
  }
```

Exercise 8.2.1 Design a tail-recursive version of `length`.

The next two functions are the complements of `head` and `tail`.

```
  def last: a
  def init: List[a]
```

`xs.last` returns the last element of list `xs`, whereas `xs.init` returns all elements of `xs` except the last. Both functions have to traverse the entire list, and are thus less efficient than their `head` and `tail` analogues. Here is the implementation of `last`.

```
  def last: a = this match {
    case Nil      => throw new Error("Nil.last")
    case x :: Nil => x
    case x :: xs  => xs.last
  }
```

The implementation of `init` is analogous.

The next three functions return a prefix of the list, or a suffix, or both.

```
  def take(n: int): List[a] =
    if (n == 0 || isEmpty) Nil else head :: tail.take(n-1)

  def drop(n: int): List[a] =
    if (n == 0 || isEmpty) this else tail.drop(n-1)

  def split(n: int): {List[a], List[a]} = {take(n), drop(n)}
```

`(xs take n)` returns the first `n` elements of list `xs`, or the whole list, if its length is smaller than `n`. `(xs drop n)` returns all elements of `xs` except the `n` first ones. Finally, `(xs split n)` returns a pair consisting of the lists resulting from `xs take n` and `xs drop n`.

The next function returns an element at a given index in a list. It is thus analogous to array subscripting. Indices start at 0.

```
def apply(n: int): a = drop(n).head
```

The `apply` method has a special meaning in Scala. An object with an `apply` method can be applied to arguments as if it was a function. For instance, to pick the 3'rd element of a list `xs`, one can write either `xs.apply(3)` or `xs(3)` – the latter expression expands into the first.

With `take` and `drop`, we can extract sublists consisting of consecutive elements of the original list. To extract the sublist $xs_m, \ldots, xs_{n-1}$ of a list xs, use:

```
xs.drop(m).take(n - m)
```

Zipping lists. The next function combines two lists into a list of pairs. Given two lists

xs = List(x_1, ..., x_n) , and
ys = List(y_1, ..., y_n) ,

`xs zip ys` constructs the list List({x_1, y_1}, ..., {x_n, y_n}). If the two lists have different lengths, the longer one of the two is truncated. Here is the definition of `zip` – note that it is a polymorphic method.

```
def zip[b](that: List[b]): List[{a,b}] =
  if (this.isEmpty || that.isEmpty) Nil
  else {this.head, that.head} :: (this.tail zip that.tail)
```

Consing lists.. Like any infix operator, `::` is also implemented as a method of an object. In this case, the object is the list that is extended. This is possible, because operators ending with a '`:`' character are treated specially in Scala. All such operators are treated as methods of their right operand. E.g.,

`x :: y = y.::(x)` whereas `x + y = x.+(y)`

Note, however, that operands of a binary operation are in each case evaluated from left to right. So, if `D` and `E` are expressions with possible side-effects, `D :: E` is translated to `{val x = D; E.::(x)}` in order to maintain the left-to-right order of operand evaluation.

Another difference between operators ending in a '`:`' and other operators concerns their associativity. Operators ending in '`:`' are right-associative, whereas other operators are left-associative. E.g.,

`x :: y :: z = x :: (y :: z)` whereas `x + y + z = (x + y) + z`

The definition of `::` as a method in class `List` is as follows:

```
def ::[b >: a](x: b): List[b] = new scala.::(x, this)
```

Note that `::` is defined for all elements `x` of type `B` and lists of type `List[A]` such that the type `B` of `x` is a supertype of the list's element type `A`. The result is in this case a list of `B`'s. This is expressed by the type parameter `b` with lower bound `a` in the signature of `::`.

Concatenating lists. An operation similar to `::` is list concatenation, written '`:::`'. The result of `(xs ::: ys)` is a list consisting of all elements of `xs`, followed by all elements of `ys`. Because it ends in a colon, `:::` is right-associative and is considered as a method of its right-hand operand. Therefore,

```
xs ::: ys ::: zs  =   xs ::: (ys ::: zs)
                  =   zs.:::(ys).:::(xs)
```

Here is the implementation of the `:::` method:

```
  def :::[b >: a](prefix: List[b]): List[b] = prefix match {
    case Nil => this
    case p :: ps => this.:::(ps).::(p)
  }
```

Reversing lists. Another useful operation is list reversal. There is a method `reverse` in `List` to that effect. Let's try to give its implementation:

```
def reverse[a](xs: List[a]): List[a] = xs match {
  case Nil => Nil
  case x :: xs => reverse(xs) ::: List(x)
}
```

This implementation has the advantage of being simple, but it is not very efficient. Indeed, one concatenation is executed for every element in the list. List concatenation takes time proportional to the length of its first operand. Therefore, the complexity of `reverse(xs)` is

$$n + (n-1) + ... + 1 = n(n+1)/2$$

where n is the length of `xs`. Can `reverse` be implemented more efficiently? We will see later that there exists another implementation which has only linear complexity.

8.3 Example: Merge sort

The insertion sort presented earlier in this chapter is simple to formulate, but also not very efficient. It's average complexity is proportional to the square of the length of the input list. We now design a program to sort the elements of a list which is more efficient than insertion sort. A good algorithm for this is *merge sort*, which works as follows.

First, if the list has zero or one elements, it is already sorted, so one returns the list unchanged. Longer lists are split into two sub-lists, each containing about half the elements of the original list. Each sub-list is sorted by a recursive call to the sort function, and the resulting two sorted lists are then combined in a merge operation.

For a general implementation of merge sort, we still have to specify the type of list elements to be sorted, as well as the function to be used for the comparison of elements. We obtain a function of maximal generality by passing these two items as parameters. This leads to the following implementation.

```
def msort[a](less: (a, a) => boolean)(xs: List[a]): List[a] = {
  def merge(xs1: List[a], xs2: List[a]): List[a] =
    if (xs1.isEmpty) xs2
    else if (xs2.isEmpty) xs1
    else if (less(xs1.head, xs2.head)) xs1.head :: merge(xs1.tail, xs2)
    else xs2.head :: merge(xs1, xs2.tail)
  val n = xs.length/2
  if (n == 0) xs
  else merge(msort(less)(xs take n), msort(less)(xs drop n))
}
```

The complexity of `msort` is $O(N\ log(N))$, where N is the length of the input list. To see why, note that splitting a list in two and merging two sorted lists each take time proportional to the length of the argument list(s). Each recursive call of `msort` halves the number of elements in its input, so there are $O(log(N))$ consecutive recursive calls until the base case of lists of length 1 is reached. However, for longer lists each call spawns off two further calls. Adding everything up we obtain that at each of the $O(log(N))$ call levels, every element of the original lists takes part in one split operation and in one merge operation. Hence, every call level has a total cost proportional to $O(N)$. Since there are $O(log(N))$ call levels, we obtain an overall cost of $O(N\ log(N))$. That cost does not depend on the initial distribution of elements in the list, so the worst case cost is the same as the average case cost. This makes merge sort an attractive algorithm for sorting lists.

Here is an example how `msort` is used.

```
msort(x: int, y: int => x < y)(List(5, 7, 1, 3))
```

The definition of `msort` is curried, to make it easy to specialize it with particular comparison functions. For instance,

```
val intSort = msort(x: int, y: int => x < y)
val reverseSort = msort(x: int, y: int => x > y)
```

8.4 Definition of class List II: Higher-Order Methods

The examples encountered so far show that functions over lists often have similar structures. We can identify several patterns of computation over lists, like:

- transforming every element of a list in some way.
- extracting from a list all elements satisfying a criterion.
- combine the elements of a list using some operator.

Functional programming languages enable programmers to write general functions which implement patterns like this by means of higher order functions. We now discuss a set of commonly used higher-order functions, which are implemented as methods in class `List`.

Mapping over lists. A common operation is to transform each element of a list and then return the lists of results. For instance, to scale each element of a list by a given factor.

```
def scaleList(xs: List[double], factor: double): List[double] = xs match {
  case Nil => xs
  case x :: xs1 => x * factor :: scaleList(xs1, factor)
}
```

This pattern can be generalized to the `map` method of class `List`:

```
abstract class List[a] { ...
  def map[b](f: a => b): List[b] = this match {
    case Nil => this
    case x :: xs => f(x) :: xs.map(f)
  }
```

Using `map`, `scaleList` can be more concisely written as follows.

```
def scaleList(xs: List[double], factor: double) =
  xs map (x => x * factor)
```

As another example, consider the problem of returning a given column of a matrix which is represented as a list of rows, where each row is again a list. This is done by the following function `column`.

```
def column[a](xs: List[List[a[]], index: int): List[a] =
  xs map (row => row at index)
```

Closely related to `map` is the `foreach` method, which applies a given function to all elements of a list, but does not construct a list of results. The function is thus applied only for its side effect. `foreach` is defined as follows.

```
def foreach(f: a => unit) {
  this match {
    case Nil => {}
    case x :: xs => f(x) ; xs.foreach(f)
  }
}
```

This function can be used for printing all elements of a list, for instance:

```
xs foreach (x => System.out.println(x))
```

Exercise 8.4.1 Consider a function which squares all elements of a list and returns a list with the results. Complete the following two equivalent definitions of `squareList`.

```
def squareList(xs: List[int]): List[int] = xs match {
  case List() => ??
  case y :: ys => ??
}
def squareList(xs: List[int]): List[int] =
  xs map ??
```

Filtering Lists. Another common operation selects from a list all elements fulfilling a given criterion. For instance, to return a list of all positive elements in some given lists of integers:

```
def posElems(xs: List[int]): List[int] = xs match {
  case Nil => xs
  case x :: xs1 => if (x > 0) x :: posElems(xs1) else posElems(xs1)
}
```

This pattern is generalized to the `filter` method of class `List`:

```
def filter(p: a => boolean): List[a] = this match {
  case Nil => this
  case x :: xs => if (p(x)) x :: xs.filter(p) else xs.filter(p)
}
```

Using `filter`, `posElems` can be more concisely written as follows.

```
def posElems(xs: List[int]): List[int] =
  xs filter (x => x > 0)
```

An operation related to filtering is testing whether all elements of a list satisfy a certain condition. Dually, one might also be interested in the question whether there exists an element in a list that satisfies a certain condition. These operations are embodied in the higher-order functions `forall` and `exists` of class `List`.

```
def forall(p: a => boolean): boolean =
  isEmpty || (p(head) && (tail forall p))
def exists(p: a => boolean): boolean =
  !isEmpty && (p(head) || (tail exists p))
```

To illustrate the use of `forall`, consider the question whether a number if prime. Remember that a number n is prime of it can be divided without remainder only by one and itself. The most direct translation of this definition would test that n divided by all numbers from 2 up to and excluding itself gives a non-zero remainder. This list of numbers can be generated using a function `List.range` which is defined in object `List` as follows.

```
package scala
object List { ...
  def range(from: int, end: int): List[int] =
    if (from >= end) Nil else from :: range(from + 1, end)
```

For example, `List.range(2, n)` generates the list of all integers from 2 up to and excluding n. The function `isPrime` can now simply be defined as follows.

```
def isPrime(n: int) =
  List.range(2, n) forall (x => n % x != 0)
```

We see that the mathematical definition of prime-ness has been translated directly into Scala code.

Exercise: Define `forall` and `exists` in terms of `filter`.

Folding and Reducing Lists. Another common operation is to combine the elements of a list with some operator. For instance:

$$\begin{array}{lcl}
\texttt{sum(List(}x_1\texttt{, ..., }x_n\texttt{))} & = & \texttt{0 + }x_1\texttt{ + ... + }x_n \\
\texttt{product(List(}x_1\texttt{, ..., }x_n\texttt{))} & = & \texttt{1 * }x_1\texttt{ * ... * }x_n
\end{array}$$

Of course, we can implement both functions with a recursive scheme:

```
def sum(xs: List[int]): int = xs match {
  case Nil => 0
  case y :: ys => y + sum(ys)
}
def product(xs: List[int]): int = xs match {
  case Nil => 1
  case y :: ys => y * product(ys)
}
```

But we can also use the generalization of this program scheme embodied in the `reduceLeft` method of class `List`. This method inserts a given binary operator between adjacent elements of a given list. E.g.

```
List(x1, ..., xn).reduceLeft(op) = (...(x1 op x2) op ... ) op xn
```

Using `reduceLeft`, we can make the common pattern in `sum` and `product` apparent:

```
def sum(xs: List[int])      =  (0 :: xs) reduceLeft {(x, y) => x + y}
def product(xs: List[int])  =  (1 :: xs) reduceLeft {(x, y) => x * y}
```

Here is the implementation of `reduceLeft`.

```
    def reduceLeft(op: (a, a) => a): a = this match {
      case Nil     => throw new Error("Nil.reduceLeft")
      case x :: xs => (xs foldLeft x)(op)
    }
    def foldLeft[b](z: b)(op: (b, a) => b): b = this match {
      case Nil => z
      case x :: xs => (xs foldLeft op(z, x))(op)
    }
  }
```

We see that the `reduceLeft` method is defined in terms of another generally useful method, `foldLeft`. The latter takes as additional parameter an *accumulator* `z`, which is returned when `foldLeft` is applied on an empty list. That is,

```
(List(x1, ..., xn) foldLeft z)(op)   =  (...(z op x1) op ... ) op xn
```

The `sum` and `product` methods can be defined alternatively using `foldLeft`:

```
def sum(xs: List[int])      =  (xs foldLeft 0) {(x, y) => x + y}
def product(xs: List[int])  =  (xs foldLeft 1) {(x, y) => x * y}
```

FoldRight and ReduceRight. Applications of `foldLeft` and `reduceLeft` expand to left-leaning trees. . They have duals `foldRight` and `reduceRight`, which produce right-leaning trees.

```
List(x1, ..., xn).reduceRight(op)     =  x1 op ( ... (xn-1 op xn)...)
(List(x1, ..., xn) foldRight acc)(op) =  x1 op ( ... (xn op acc)...)
```

These are defined as follows.

```
    def reduceRight(op: (a, a) => a): a = this match {
      case Nil => throw new Error("Nil.reduceRight")
      case x :: Nil => x
      case x :: xs => op(x, xs.reduceRight(op))
    }
    def foldRight[b](z: b)(op: (a, b) => b): b = this match {
      case Nil => z
      case x :: xs => op(x, (xs foldRight z)(op))
    }
```

Class `List` defines also two symbolic abbreviations for `foldLeft` and `foldRight`:

```
    def /:[b](z: b)(f: (b, a) => b): b = foldLeft(z)(f)
    def :\[b](z: b)(f: (a, b) => b): b = foldRight(z)(f)
```

The method names picture the left/right leaning trees of the fold operations by forward or backward slashes. The `:` points in each case to the list argument whereas the end of the slash points to the accumulator (or: zero) argument `z`. That is,

```
  (z /: List(x1, ..., xn))(op) = (...(z op x1) op ... ) op xn
  (List(x1, ..., xn) :\ z)(op) = x1 op ( ... (xn op acc)...)
```

For associative and commutative operators, `/:` and `:\` are equivalent (even though there may be a difference in efficiency).

Exercise 8.4.2 Consider the problem of writing a function `flatten`, which takes a list of element lists as arguments. The result of `flatten` should be the concatenation of all element lists into a single list. Here is the an implementation of this method in terms of `:\`.

```
  def flatten[a](xs: List[List[a]]): List[a] =
    (xs :\ (Nil: List[a])) {(x, xs) => x ::: xs}
```

Consider replacing the body of `flatten` by

```
    ((Nil: List[a]) /: xs) ((xs, x) => xs ::: x)
```

What would be the difference in asymptotic complexity between the two versions of `flatten`?

In fact `flatten` is predefined together with a set of other userful function in an object called `List` in the standatd Scala library. It can be accessed from user program by calling `List.flatten`. Note that `flatten` is not a method of class `List` – it would not make sense there, since it applies only to lists of lists, not to all lists in general.

List Reversal Again. We have seen in Section 8.2 an implementation of method `reverse` whose run-time was quadratic in the length of the list to be reversed. We now develop a new implementation of `reverse`, which has linear cost. The idea is to use a `foldLeft` operation based on the following program scheme.

```
  class List[+a] { ...
```

```
    def reverse: List[a] = (z? /: this)(op?)
```

It only remains to fill in the z? and op? parts. Let's try to deduce them from examples.

```
    Nil
  = Nil.reverse                // by specification
  = (z /: Nil)(op)             // by the template for reverse
  = (Nil foldLeft z)(op)       // by the definition of /:
  = z                          // by definition of foldLeft
```

Hence, z? must be Nil. To deduce the second operand, let's study reversal of a list of length one.

```
    List(x)
  = List(x).reverse            // by specification
  = (Nil /: List(x))(op)       // by the template for reverse, with z = Nil
  = (List(x) foldLeft Nil)(op) // by the definition of /:
  = op(Nil, x)                 // by definition of foldLeft
```

Hence, op(Nil, x) equals List(x), which is the same as x :: Nil. This suggests to take as op the :: operator with its operands exchanged. Hence, we arrive at the following implementation for reverse, which has linear complexity.

```
  def reverse: List[a] =
    ((Nil: List[a]) /: this) {(xs, x) => x :: xs}
```

(Remark: The type annotation of Nil is necessary to make the type inferencer work.)

Exercise 8.4.3 Fill in the missing expressions to complete the following definitions of some basic list-manipulation operations as fold operations.

```
  def mapFun[a, b](xs: List[a], f: a => b): List[b] =
    (xs :\ List[b]()){ ?? }

  def lengthFun[a](xs: List[a]): int =
    (0 /: xs){ ?? }
```

Nested Mappings. We can employ higher-order list processing functions to express many computations that are normally expressed as nested loops in imperative languages.

As an example, consider the following problem: Given a positive integer n, find all pairs of positive integers i and j, where $1 \leq j < i < n$ such that $i + j$ is prime. For instance, if $n = 7$, the pairs are

i	2	3	4	4	5	6	6
j	1	2	1	3	2	1	5
$i+j$	3	5	5	7	7	7	11

A natural way to solve this problem consists of two steps. In a first step, one generates the sequence of all pairs $\{i, j\}$ of integers such that $1 \le j < i < n$. In a second step one then filters from this sequence all pairs $\{i, j\}$ such that $i + j$ is prime.

Looking at the first step in more detail, a natural way to generate the sequence of pairs consists of three sub-steps. First, generate all integers between 1 and n for i.

Second, for each integer i between 1 and n, generate the list of pairs $\{i, 1\}$ up to $\{i, i-1\}$. This can be achieved by a combination of `range` and `map`:

```
List.range(1, i) map (x => {i, x})
```

Finally, combine all sublists using `foldRight` with `:::`. Putting everything together gives the following expression:

```
List.range(1, n)
  .map(i => List.range(1, i).map(x => {i, x}))
  .foldRight(List[{int, int}]()) {(xs, ys) => xs ::: ys}
  .filter(pair => isPrime(pair._1 + pair._2))
```

Flattening Maps. The combination of mapping and then concatenating sublists resulting from the map is so common that we there is a special method for it in class `List`:

```
abstract class List[+a] { ...
  def flatMap[b](f: a => List[b]): List[b] = this match {
    case Nil => Nil
    case x :: xs => f(x) ::: (xs flatMap f)
  }
}
```

With `flatMap`, the pairs-whose-sum-is-prime expression could have been written more concisely as follows.

```
List.range(1, n)
  .flatMap(i => List.range(1, i).map(x => {i, x}))
  .filter(pair => isPrime(pair._1 + pair._2))
```

8.5 Summary

This chapter has ingtroduced lists as a fundamental data structure in programming. Since lists are immutable, they are a common data type in functional programming languages. They play there a role comparable to arrays in imperative languages. However, the access patterns between arrays and lists are quite different. Where array accessing is always done by indexing, this is much less common for lists. We have seen that `scala.List` defines a method called `apply` for indexing however this operation is much more costly than in the case of arrays (linear as opposed to constant time). Instead of indexing, lists are usually traversed recursively, where recursion steps are usually based on a pattern match over the traversed list. There is also a rich set of higher-order combinators which allow one to instantiate a set of predefined patterns of computations over lists.

Chapter 9

For-Comprehensions

The last chapter demonstrated that higher-order functions such as `map`, `flatMap`, `filter` provide powerful constructions for dealing with lists. But sometimes the level of abstraction required by these functions makes a program hard to understand.

To help understandability, Scala has a special notation which simplifies common patterns of applications of higher-order functions. This notation builds a bridge between set-comprehensions in mathematics and for-loops in imperative languages such as C or Java. It also closely resembles the query notation of relational databases.

As a first example, say we are given a list `persons` of persons with `name` and `age` fields. To print the names of all persons in the sequence which are aged over 20, one can write:

```
for (p <- persons if p.age > 20) yield p.name
```

This is equivalent to the following expression , which uses higher-order functions `filter` and `map`:

```
persons filter (p => p.age > 20) map (p => p.name)
```

The for-comprehension looks a bit like a for-loop in imperative languages, except that it constructs a list of the results of all iterations.

Generally, a for-comprehension is of the form

```
for ( s ) yield e
```

Here, *s* is a sequence of *generators*, *definitions* and *filters*. A *generator* is of the form `val x <- e`, where `e` is a list-valued expression. It binds `x` to successive values in the list. A *definition* is of the form `val x = e`. It introduces `x` as a name for the value of `e` in the rest of the comprehension. A *filter* is an expression `f` of type `boolean`. It omits from consideration all bindings for which `f` is **`false`**. The sequence *s* starts in each case with a generator. If there are several generators in a sequence, later generators vary more rapidly than earlier ones.

The sequence *s* may also be enclosed in braces instead of parentheses, in which case the semicolons between generators, definitions and filters can be omitted.

Here are two examples that show how for-comprehensions are used. First, let's redo an example of the previous chapter: Given a positive integer n, find all pairs of positive integers i and j, where $1 \leq j < i < n$ such that $i + j$ is prime. With a for-comprehension this problem is solved as follows:

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

This is arguably much clearer than the solution using `map`, `flatMap` and `filter` that we have developed previously.

As a second example, consider computing the scalar product of two vectors `xs` and `ys`. Using a for-comprehension, this can be written as follows.

```
sum (for((x, y) <- xs zip ys) yield x * y)
```

9.1 The N-Queens Problem

For-comprehensions are especially useful for solving combinatorial puzzles. An example of such a puzzle is the 8-queens problem: Given a standard chess-board, place 8 queens such that no queen is in check from any other (a queen can check another piece if they are on the same column, row, or diagonal). We will now develop a solution to this problem, generalizing it to chess-boards of arbitrary size. Hence, the problem is to place n queens on a chess-board of size $n \times n$.

To solve this problem, note that we need to place a queen in each row. So we could place queens in successive rows, each time checking that a newly placed queen is not in check from any other queens that have already been placed. In the course of this search, it might arrive that a queen to be placed in row k would be in check in all fields of that row from queens in row 1 to $k-1$. In that case, we need to abort that part of the search in order to continue with a different configuration of queens in columns 1 to $k-1$.

This suggests a recursive algorithm. Assume that we have already generated all solutions of placing $k-1$ queens on a board of size $n \times n$. We can represent each such solution by a list of length $k-1$ of column numbers (which can range from 1 to n). We treat these partial solution lists as stacks, where the column number of the queen in row $k-1$ comes first in the list, followed by the column number of the queen in row $k-2$, etc. The bottom of the stack is the column number of the queen placed in the first row of the board. All solutions together are then represented as a list of lists, with one element for each solution.

Now, to place the *k*'the queen, we generate all possible extensions of each previous solution by one more queen. This yields another list of solution lists, this time of length *k*. We continue the process until we have reached solutions of the size of the chess-board *n*. This algorithmic idea is embodied in function placeQueens below:

```
def queens(n: int): List[List[int]] = {
  def placeQueens(k: int): List[List[int]] =
    if (k == 0) List(List())
    else for { queens <- placeQueens(k - 1)
               column <- List.range(1, n + 1)
               if isSafe(column, queens, 1) } yield column :: queens
  placeQueens(n)
}
```

Exercise 9.1.1 Write the function

```
  def isSafe(col: int, queens: List[int], delta: int): boolean
```

which tests whether a queen in the given column col is safe with respect to the queens already placed. Here, delta is the difference between the row of the queen to be placed and the row of the first queen in the list.

9.2 Querying with For-Comprehensions

The for-notation is essentially equivalent to common operations of database query languages. For instance, say we are given a database books, represented as a list of books, where Book is defined as follows.

```
case class Book(title: String, authors: List[String])
```

Here is a small example database:

```
val books: List[Book] = List(
  Book("Structure and Interpretation of Computer Programs",
       List("Abelson, Harold", "Sussman, Gerald J.")),
  Book("Principles of Compiler Design",
       List("Aho, Alfred", "Ullman, Jeffrey")),
  Book("Programming in Modula-2",
       List("Wirth, Niklaus")),
  Book("Introduction to Functional Programming"),
       List("Bird, Richard")),
  Book("The Java Language Specification",
       List("Gosling, James", "Joy, Bill", "Steele, Guy", "Bracha, Gilad")))
```

Then, to find the titles of all books whose author's last name is "Ullman":

```
for (b <- books; a <- b.authors if a startsWith "Ullman")
yield b.title
```

(Here, `startsWith` is a method in `java.lang.String`). Or, to find the titles of all books that have the string "Program" in their title:

```
for (b <- books if (b.title indexOf "Program") >= 0)
yield b.title
```

Or, to find the names of all authors that have written at least two books in the database.

```
for (b1 <- books; b2 <- books if b1 != b2;
     a1 <- b1.authors; a2 <- b2.authors if a1 == a2)
yield a1
```

The last solution is not yet perfect, because authors will appear several times in the list of results. We still need to remove duplicate authors from result lists. This can be achieved with the following function.

```
def removeDuplicates[a](xs: List[a]): List[a] =
  if (xs.isEmpty) xs
  else xs.head :: removeDuplicates(xs.tail filter (x => x != xs.head))
```

Note that the last expression in method `removeDuplicates` can be equivalently expressed using a for-comprehension.

```
xs.head :: removeDuplicates(for (x <- xs.tail if x != xs.head) yield x)
```

9.3 Translation of For-Comprehensions

Every for-comprehension can be expressed in terms of the three higher-order functions `map`, `flatMap` and `filter`. Here is the translation scheme, which is also used by the Scala compiler.

- A simple for-comprehension

  ```
  for (x <- e) yield e'
  ```

  is translated to

  ```
  e.map(x => e')
  ```

- A for-comprehension

```
for (x <- e if f; s) yield e'
```

where f is a filter and s is a (possibly empty) sequence of generators or filters is translated to

```
for (x <- e.filter(x => f); s) yield e'
```

and then translation continues with the latter expression.

- A for-comprehension

  ```
  for (x <- e; y <- e'; s) yield e''
  ```

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

  ```
  e.flatMap(x => for (y <- e'; s) yield e'')
  ```

  and then translation continues with the latter expression.

For instance, taking our "pairs of integers whose sum is prime" example:

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

Here is what we get when we translate this expression:

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

Conversely, it would also be possible to express functions map, flatMap and filter using for-comprehensions. Here are the three functions again, this time implemented using for-comprehensions.

```
object Demo {
  def map[a, b](xs: List[a], f: a => b): List[b] =
    for (x <- xs) yield f(x)

  def flatMap[a, b](xs: List[a], f: a => List[b]): List[b] =
    for (x <- xs; y <- f(x)) yield y

  def filter[a](xs: List[a], p: a => boolean): List[a] =
    for (x <- xs if p(x)) yield x
}
```

Not surprisingly, the translation of the for-comprehension in the body of `Demo.map` will produce a call to `map` in class `List`. Similarly, `Demo.flatMap` and `Demo.filter` translate to `flatMap` and `filter` in class `List`.

Exercise 9.3.1 Define the following function in terms of **for**.

```
def flatten[a](xss: List[List[a]]): List[a] =
  (xss :\ (Nil: List[a])) ((xs, ys) => xs ::: ys)
```

Exercise 9.3.2 Translate

```
for ( val b <- books; val a <- b.authors; a startsWith "Bird" ) yield b.title
for ( val b <- books; (b.title indexOf "Program") >= 0 ) yield b.title
```

to higher-order functions.

9.4 For-Loops

For-comprehensions resemble for-loops in imperative languages, except that they produce a list of results. Sometimes, a list of results is not needed but we would still like the flexibility of generators and filters in iterations over lists. This is made possible by a variant of the for-comprehension syntax, which expresses for-loops:

for (*s*) *e*

This construct is the same as the standard for-comprehension syntax except that the keyword **yield** is missing. The for-loop is executed by executing the expression *e* for each element generated from the sequence of generators and filters *s*.

As an example, the following expression prints out all elements of a matrix represented as a list of lists:

```
for (xs <- xss) {
  for (x <- xs) System.out.print(x + "\t")
  System.out.println()
}
```

The translation of for-loops to higher-order methods of class `List` is similar to the translation of for-comprehensions, but is simpler. Where for-comprehensions translate to `map` and `flatMap`, for-loops translate in each case to `foreach`.

9.5 Generalizing For

We have seen that the translation of for-comprehensions only relies on the presence of methods `map`, `flatMap`, and `filter`. Therefore it is possible to apply the same notation to generators that produce objects other than lists; these objects only have to support the three key functions `map`, `flatMap`, and `filter`.

The standard Scala library has several other abstractions that support these three methods and with them support for-comprehensions. We will encounter some of them in the following chapters. As a programmer you can also use this principle to enable for-comprehensions for types you define – these types just need to support methods `map`, `flatMap`, and `filter`.

There are many examples where this is useful: Examples are database interfaces, XML trees, or optional values. We will see in Chapter 14.2 how for-comprehensions can be used in the definition of parsers for context-free grammars that construct abstract syntax trees.

One caveat: It is not assured automatically that the result translating a for-comprehension is well-typed. To ensure this, the types of `map`, `flatMap` and `filter` have to be essentially similar to the types of these methods in class `List`.

To make this precise, assume you have a parameterized class `C[a]` for which you want to enable for-comprehensions. Then `C` should define `map`, `flatMap` and `filter` with the following types:

```
def map[b](f: a => b): C[b]
def flatMap[b](f: a => C[b]): C[b]
def filter(p: a => boolean): C[a]
```

It would be attractive to enforce these types statically in the Scala compiler, for instance by requiring that any type supporting for-comprehensions implements a standard trait with these methods [1]. The problem is that such a standard trait would have to abstract over the identity of the class `C`, for instance by taking `C` as a type parameter. Note that this parameter would be a type constructor, which gets applied to *several different* types in the signatures of methods `map` and `flatMap`. Unfortunately, the Scala type system is too weak to express this construct, since it can handle only type parameters which are fully applied types.

[1] In the programming language Haskell, which has similar constructs, this abstraction is called a "monad with zero"

Chapter 10

Mutable State

Most programs we have presented so far did not have side-effects [1]. Therefore, the notion of *time* did not matter. For a program that terminates, any sequence of actions would have led to the same result! This is also reflected by the substitution model of computation, where a rewrite step can be applied anywhere in a term, and all rewritings that terminate lead to the same solution. In fact, this *confluence* property is a deep result in λ-calculus, the theory underlying functional programming.

In this chapter, we introduce functions with side effects and study their behavior. We will see that as a consequence we have to fundamentally modify up the substitution model of computation which we employed so far.

10.1 Stateful Objects

We normally view the world as a set of objects, some of which have state that *changes* over time. Normally, state is associated with a set of variables that can be changed in the course of a computation. There is also a more abstract notion of state, which does not refer to particular constructs of a programming language: An object *has state* (or: *is stateful*) if its behavior is influenced by its history.

For instance, a bank account object has state, because the question "can I withdraw 100 CHF?" might have different answers during the lifetime of the account.

In Scala, all mutable state is ultimately built from variables. A variable definition is written like a value definition, but starts with `var` instead of `val`. For instance, the following two definitions introduce and initialize two variables `x` and `count`.

```
  var x: String = "abc"
```

[1]We ignore here the fact that some of our program printed to standard output, which technically is a side effect.

```
var count = 111
```

Like a value definition, a variable definition associates a name with a value. But in the case of a variable definition, this association may be changed later by an assignment. Such assignments are written as in C or Java. Examples:

```
x = "hello"
count = count + 1
```

In Scala, every defined variable has to be initialized at the point of its definition. For instance, the statement **var** x: int; is *not* regarded as a variable definition, because the initializer is missing[2]. If one does not know, or does not care about, the appropriate initializer, one can use a wildcard instead. I.e.

```
val x: T = _
```

will initialize x to some default value (**null** for reference types, **false** for booleans, and the appropriate version of 0 for numeric value types).

Real-world objects with state are represented in Scala by objects that have variables as members. For instance, here is a class that represents bank accounts.

```
class BankAccount {
  private var balance = 0
  def deposit(amount: int) {
    if (amount > 0) balance = balance + amount
  }

  def withdraw(amount: int): int =
    if (0 < amount && amount <= balance) {
      balance = balance - amount
      balance
    } else throw new Error("insufficient funds")
}
```

The class defines a variable balance which contains the current balance of an account. Methods deposit and withdraw change the value of this variable through assignments. Note that balance is **private** in class BankAccount – hence it can not be accessed directly outside the class.

To create bank-accounts, we use the usual object creation notation:

```
val myAccount = new BankAccount
```

[2]If a statement like this appears in a class, it is instead regarded as a variable declaration, which introduces abstract access methods for the variable, but does not associate these methods with a piece of state.

Example 10.1.1 Here is a `scalaint` session that deals with bank accounts.

```
> :l bankaccount.scala
loading file 'bankaccount.scala'
> val account = new BankAccount
val account : BankAccount = BankAccount$class@1797795
> account deposit 50
{}: scala.Unit
> account withdraw 20
30: scala.Int
> account withdraw 20
10: scala.Int
> account withdraw 15
java.lang.RuntimeException: insufficient funds
        at BankAccount$class.withdraw(bankaccount.scala:13)
        at <top-level>(console:1)
>
```

The example shows that applying the same operation (`withdraw 20`) twice to an account yields different results. So, clearly, accounts are stateful objects.

Sameness and Change. Assignments pose new problems in deciding when two expressions are "the same". If assignments are excluded, and one writes

```
val x = E; val y = E
```

where `E` is some arbitrary expression, then `x` and `y` can reasonably be assumed to be the same. I.e. one could have equivalently written

```
val x = E; val y = x
```

(This property is usually called *referential transparency*). But once we admit assignments, the two definition sequences are different. Consider:

```
val x = new BankAccount; val y = new BankAccount
```

To answer the question whether `x` and `y` are the same, we need to be more precise what "sameness" means. This meaning is captured in the notion of *operational equivalence*, which, somewhat informally, is stated as follows.

Suppose we have two definitions of `x` and `y`. To test whether `x` and `y` define the same value, proceed as follows.

- Execute the definitions followed by an arbitrary sequence `S` of operations that involve `x` and `y`. Observe the results (if any).
- Then, execute the definitions with another sequence `S'` which results from `S` by renaming all occurrences of `y` in `S` to `x`.

- If the results of running `S'` are different, then surely x and y are different.
- On the other hand, if all possible pairs of sequences `{S, S'}` yield the same results, then x and y are the same.

In other words, operational equivalence regards two definitions x and y as defining the same value, if no possible experiment can distinguish between x and y. An experiment in this context are two version of an arbitrary program which use either x or y.

Given this definition, let's test whether

```
val x = new BankAccount; val y = new BankAccount
```

defines values x and y which are the same. Here are the definitions again, followed by a test sequence:

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> y withdraw 20
java.lang.RuntimeException: insufficient funds
```

Now, rename all occurrences of y in that sequence to x. We get:

```
> val x = new BankAccount
> val y = new BankAccount
> x deposit 30
30
> x withdraw 20
10
```

Since the final results are different, we have established that x and y are not the same. On the other hand, if we define

```
val x = new BankAccount; val y = x
```

then no sequence of operations can distinguish between x and y, so x and y are the same in this case.

Assignment and the Substitution Model. These examples show that our previous substitution model of computation cannot be used anymore. After all, under this model we could always replace a value name by its defining expression. For instance in

```
val x = new BankAccount; val y = x
```

the x in the definition of y could be replaced by **new** BankAccount. But we have seen that this change leads to a different program. So the substitution model must be invalid, once we add assignments.

10.2 Imperative Control Structures

Scala has the **while** and **do-while** loop constructs known from the C and Java languages. There is also a single branch **if** which leaves out the else-part as well as a **return** statement which aborts a function prematurely. This makes it possible to program in a conventional imperative style. For instance, the following function, which computes the n'th power of a given parameter x, is implemented using **while** and single-branch **if**.

```
def power(x: double, n: int): double = {
  var r = 1.0
  var i = n
  var j = 0
  while (j < 32) {
    r = r * r
    if (i < 0)
      r = r * x
    i = i << 1
    j = j + 1
  }
  r
}
```

These imperative control constructs are in the language for convenience. They could have been left out, as the same constructs can be implemented using just functions. As an example, let's develop a functional implementation of the while loop. whileLoop should be a function that takes two parameters: a condition, of type boolean, and a command, of type unit. Both condition and command need to be passed by-name, so that they are evaluated repeatedly for each loop iteration. This leads to the following definition of whileLoop.

```
def whileLoop(condition: => boolean)(command: => unit): unit =
  if (condition) {
    command; whileLoop(condition)(command)
  } else {}
```

Note that whileLoop is tail recursive, so it operates in constant stack space.

Exercise 10.2.1 Write a function repeatLoop, which should be applied as follows:

```
repeatLoop { command } ( condition )
```

Is there also a way to obtain a loop syntax like the following?

```
repeatLoop { command } until ( condition )
```

Some other control constructs known from C and Java are missing in Scala: There are no `break` and `continue` jumps for loops. There are also no for-loops in the Java sense – these have been replaced by the more general for-loop construct discussed in Section 9.4.

10.3 Extended Example: Discrete Event Simulation

We now discuss an example that demonstrates how assignments and higher-order functions can be combined in interesting ways. We will build a simulator for digital circuits.

The example is taken from Abelson and Sussman's book [ASS96]. We augment their basic (Scheme-) code by an object-oriented structure which allows code-reuse through inheritance. The example also shows how discrete event simulation programs in general are structured and built.

We start with a little language to describe digital circuits. A digital circuit is built from *wires* and *function boxes*. Wires carry signals which are transformed by function boxes. We will represent signals by the booleans **`true`** and **`false`**.

Basic function boxes (or: *gates*) are:

- An *inverter*, which negates its signal
- An *and-gate*, which sets its output to the conjunction of its input.
- An *or-gate*, which sets its output to the disjunction of its input.

Other function boxes can be built by combining basic ones.

Gates have *delays*, so an output of a gate will change only some time after its inputs change.

A Language for Digital Circuits. We describe the elements of a digital circuit by the following set of Scala classes and functions.

First, there is a class `Wire` for wires. We can construct wires as follows.

```
val a = new Wire
val b = new Wire
val c = new Wire
```

Second, there are procedures

```
def inverter(input: Wire, output: Wire)
def andGate(a1: Wire, a2: Wire, output: Wire)
def orGate(o1: Wire, o2: Wire, output: Wire)
```

which "make" the basic gates we need (as side-effects). More complicated function boxes can now be built from these. For instance, to construct a half-adder, we can define:

```
  def halfAdder(a: Wire, b: Wire, s: Wire, c: Wire) {
    val d = new Wire
    val e = new Wire
    orGate(a, b, d)
    andGate(a, b, c)
    inverter(c, e)
    andGate(d, e, s)
  }
```

This abstraction can itself be used, for instance in defining a full adder:

```
  def fullAdder(a: Wire, b: Wire, cin: Wire, sum: Wire, cout: Wire) {
    val s = new Wire
    val c1 = new Wire
    val c2 = new Wire
    halfAdder(a, cin, s, c1)
    halfAdder(b, s, sum, c2)
    orGate(c1, c2, cout)
  }
```

Class `Wire` and functions `inverter`, `andGate`, and `orGate` represent thus a little language in which users can define digital circuits. We now give implementations of this class and these functions, which allow one to simulate circuits. These implementations are based on a simple and general API for discrete event simulation.

The Simulation API. Discrete event simulation performs user-defined *actions* at specified *times*. An *action* is represented as a function which takes no parameters and returns a `unit` result:

```
type Action = () => unit
```

The *time* is simulated; it is not the actual "wall-clock" time.

A concrete simulation will be done inside an object which inherits from the abstract `Simulation` class. This class has the following signature:

```
abstract class Simulation {
  def currentTime: int
  def afterDelay(delay: int, action: => Action)
```

```
    def run
  }
```

Here, currentTime returns the current simulated time as an integer number, afterDelay schedules an action to be performed at a specified delay after currentTime, and run runs the simulation until there are no further actions to be performed.

The Wire Class. A wire needs to support three basic actions.

> getSignal: boolean returns the current signal on the wire.
>
> setSignal(sig: boolean) sets the wire's signal to sig.
>
> addAction(p: Action) attaches the specified procedure p to the *actions* of the wire. All attached action procedures will be executed every time the signal of a wire changes.

Here is an implementation of the Wire class:

```
  class Wire {
    private var sigVal = false
    private var actions: List[Action] = List()
    def getSignal = sigVal
    def setSignal(s: boolean) =
      if (s != sigVal) {
        sigVal = s
        actions.foreach(action => action());
      }
    def addAction(a: Action) = {
      actions = a :: actions; a()
    }
  }
```

Two private variables make up the state of a wire. The variable sigVal represents the current signal, and the variable actions represents the action procedures currently attached to the wire.

The Inverter Class. We implement an inverter by installing an action on its input wire, namely the action which puts the negated input signal onto the output signal. The action needs to take effect at InverterDelay simulated time units after the input changes. This suggests the following implementation:

```
  def inverter(input: Wire, output: Wire) = {
    def invertAction() = {
      val inputSig = input.getSignal
      afterDelay(InverterDelay, () => output.setSignal(!inputSig))
    }
    input addAction invertAction
  }
```

The And-Gate Class. And-gates are implemented analogously to inverters. The action of an andGate is to output the conjunction of its input signals. This should happen at AndGateDelay simulated time units after any one of its two inputs changes. Hence, the following implementation:

```
  def andGate(a1: Wire, a2: Wire, output: Wire) = {
    def andAction() = {
      val a1Sig = a1.getSignal
      val a2Sig = a2.getSignal
      afterDelay(AndGateDelay, () => output.setSignal(a1Sig & a2Sig))
    }
    a1 addAction andAction
    a2 addAction andAction
  }
```

Exercise 10.3.1 Write the implementation of orGate.

Exercise 10.3.2 Another way is to define an or-gate by a combination of inverters and and gates. Define a function orGate in terms of andGate and inverter. What is the delay time of this function?

The Simulation Class. Now, we just need to implement class Simulation, and we are done. The idea is that we maintain inside a Simulation object an *agenda* of actions to perform. The agenda is represented as a list of pairs of actions and the times they need to be run. The agenda list is sorted, so that earlier actions come before later ones.

```
  class Simulation {
    private type Agenda = List[{int, Action}]
    private var agenda: Agenda = List()
```

There is also a private variable curtime to keep track of the current simulated time.

```
    private var curtime = 0
```

An application of the method afterDelay(delay, action) inserts the pair {curtime + delay, action} into the agenda list at the appropriate place.

```
    def afterDelay(int delay)(action: => Action) {
      val actiontime = curtime + delay
      def insertAction(ag: Agenda): Agenda = ag match {
        case List() =>
          {actiontime, action} :: ag
        case (first @ {time, act}) :: ag1 =>
          if (actiontime < time) {actiontime, action} :: ag
          else first :: insert(ag1)
      }
      agenda = insert(agenda)
    }
```

An application of the run method removes successive elements from the agenda and performs their actions. It continues until the agenda is empty:

```
  def run = {
    afterDelay(0, () => System.out.println("*** simulation started ***"))
    agenda match {
      case List() =>
      case {_, action} :: agenda1 =>
        agenda = agenda1; action(); run
    }
  }
```

Running the Simulator. To run the simulator, we still need a way to inspect changes of signals on wires. To this purpose, we write a function probe.

```
  def probe(name: String, wire: Wire) {
    wire addAction {() =>
      System.out.println(
        name + " " + currentTime + " new_value = " + wire.getSignal)
    }
  }
```

Now, to see the simulator in action, let's define four wires, and place probes on two of them:

```
  > val input1 = new Wire
  > val input2 = new Wire
  > val sum = new Wire
  > val carry = new Wire

  > probe("sum", sum)
  sum 0 new_value = false
  > probe("carry", carry)
  carry 0 new_value = false
```

Now let's define a half-adder connecting the wires:

```
> halfAdder(input1, input2, sum, carry)
```

Finally, set one after another the signals on the two input wires to **true** and run the simulation.

```
> input1 setSignal true; run
*** simulation started ***
sum 8 new_value = true
> input2 setSignal true; run
carry 11 new_value = true
sum 15 new_value = false
```

10.4 Summary

We have seen in this chapter the constructs that let us model state in Scala – these are variables, assignments, and imperative control structures. State and Assignment complicate our mental model of computation. In particular, referential transparency is lost. On the other hand, assignment gives us new ways to formulate programs elegantly. As always, it depends on the situation whether purely functional programming or programming with assignments works best.

Chapter 11

Computing with Streams

The previous chapters have introduced variables, assignment and stateful objects. We have seen how real-world objects that change with time can be modeled by changing the state of variables in a computation. Time changes in the real world thus are modeled by time changes in program execution. Of course, such time changes are usually stretched out or compressed, but their relative order is the same. This seems quite natural, but there is a also price to pay: Our simple and powerful substitution model for functional computation is no longer applicable once we introduce variables and assignment.

Is there another way? Can we model state change in the real world using only immutable functions? Taking mathematics as a guide, the answer is clearly yes: A time-changing quantity is simply modeled by a function `f(t)` with a time parameter `t`. The same can be done in computation. Instead of overwriting a variable with successive values, we represent all these values as successive elements in a list. So, a mutable variable **`var`** `x: T` gets replaced by an immutable value **`val`** `x: List[T]`. In a sense, we trade space for time – the different values of the variable now all exist concurrently as different elements of the list. One advantage of the list-based view is that we can "time-travel", i.e. view several successive values of the variable at the same time. Another advantage is that we can make use of the powerful library of list processing functions, which often simplifies computation. For instance, consider the imperative way to compute the sum of all prime numbers in an interval:

```
def sumPrimes(start: int, end: int): int = {
  var i = start
  var acc = 0
  while (i < end) {
    if (isPrime(i)) acc = acc + i
    i = i + 1
  }
  acc
}
```

Note that the variable `i` "steps through" all values of the interval `[start .. end-1]`.

A more functional way is to represent the list of values of variable `i` directly as `range(start, end)`. Then the function can be rewritten as follows.

```
def sumPrimes(start: int, end: int) =
  sum(range(start, end) filter isPrime)
```

No contest which program is shorter and clearer! However, the functional program is also considerably less efficient since it constructs a list of all numbers in the interval, and then another one for the prime numbers. Even worse from an efficiency point of view is the following example:

To find the second prime number between 1000 and 10000:

```
  range(1000, 10000) filter isPrime at 1
```

Here, the list of all numbers between 1000 and 10000 is constructed. But most of that list is never inspected!

However, we can obtain efficient execution for examples like these by a trick:

> Avoid computing the tail of a sequence unless that tail is actually necessary for the computation.

We define a new class for such sequences, which is called `Stream`.

Streams are created using the constant `empty` and the constructor `cons`, which are both defined in module `scala.Stream`. For instance, the following expression constructs a stream with elements 1 and 2:

```
Stream.cons(1, Stream.cons(2, Stream.empty))
```

As another example, here is the analogue of `List.range`, but returning a stream instead of a list:

```
def range(start: int, end: int): Stream[int] =
  if (start >= end) Stream.empty
  else Stream.cons(start, range(start + 1, end))
```

(This function is also defined as given above in module `Stream`). Even though `Stream.range` and `List.range` look similar, their execution behavior is completely different:

`Stream.range` immediately returns with a `Stream` object whose first element is `start`. All other elements are computed only when they are *demanded* by calling the `tail` method (which might be never at all).

Streams are accessed just as lists. as for lists, the basic access methods are `isEmpty`, `head` and `tail`. For instance, we can print all elements of a stream as follows.

```
def print(xs: Stream[a]) {
  if (!xs.isEmpty) { System.out.println(xs.head); print(xs.tail) }
}
```

Streams also support almost all other methods defined on lists (see below for where their methods sets differ). For instance, we can find the second prime number between `1000` and `10000` by applying methods `filter` and `apply` on an interval stream:

```
    Stream.range(1000, 10000) filter isPrime at 1
```

The difference to the previous list-based implementation is that now we do not needlessly construct and test for primality any numbers beyond 3.

Consing and appending streams. Two methods in class `List` which are not supported by class `Stream` are `::` and `:::`. The reason is that these methods are dispatched on their right-hand side argument, which means that this argument needs to be evaluated before the method is called. For instance, in the case of `x :: xs` on lists, the tail `xs` needs to be evaluated before `::` can be called and the new list can be constructed. This does not work for streams, where we require that the tail of a stream should not be evaluated until it is demanded by a `tail` operation. The argument why list-append `:::` cannot be adapted to streams is analogous.

Instead of `x :: xs`, one uses `Stream.cons(x, xs)` for constructing a stream with first element `x` and (unevaluated) rest `xs`. Instead of `xs ::: ys`, one uses the operation `xs append ys`.

Chapter 12

Iterators

Iterators are the imperative version of streams. Like streams, iterators describe potentially infinite lists. However, there is no data-structure which contains the elements of an iterator. Instead, iterators allow one to step through the sequence, using two abstract methods `next` and `hasNext`.

```
trait Iterator[+a] {
  def hasNext: boolean
  def next: a
```

Method `next` returns successive elements. Method `hasNext` indicates whether there are still more elements to be returned by `next`. Iterators also support some other methods, which are explained later.

As an example, here is an application which prints the squares of all numbers from 1 to 100.

```
var it: Iterator[int] = Iterator.range(1, 100)
while (it.hasNext) {
  val x = it.next
  System.out.println(x * x)
}
```

12.1 Iterator Methods

Iterators support a rich set of methods besides `next` and `hasNext`, which is described in the following. Many of these methods mimic a corresponding functionality in lists.

Append. Method append constructs an iterator which resumes with the given iterator it after the current iterator has finished.

```
def append[b >: a](that: Iterator[b]): Iterator[b] = new Iterator[b] {
  def hasNext = Iterator.this.hasNext || that.hasNext
  def next = if (Iterator.this.hasNext) Iterator.this.next else that.next
}
```

The terms Iterator.**this**.next and Iterator.**this**.hasNext in the definition of append call the corresponding methods as they are defined in the enclosing Iterator class. If the Iterator prefix to **this** would have been missing, hasNext and next would have called recursively the methods being defined in the result of append, which is not what we want.

Map, FlatMap, Foreach. Method map constructs an iterator which returns all elements of the original iterator transformed by a given function f.

```
def map[b](f: a => b): Iterator[b] = new Iterator[b] {
  def hasNext = Iterator.this.hasNext
  def next = f(Iterator.this.next)
}
```

Method flatMap is like method map, except that the transformation function f now returns an iterator. The result of flatMap is the iterator resulting from appending together all iterators returned from successive calls of f.

```
def flatMap[b](f: a => Iterator[b]): Iterator[b] = new Iterator[b] {
  private var cur: Iterator[b] = Iterator.empty
  def hasNext: boolean =
    if (cur.hasNext) true
    else if (Iterator.this.hasNext) { cur = f(Iterator.this.next); hasNext }
    else false
  def next: b =
    if (cur.hasNext) cur.next
    else if (Iterator.this.hasNext) { cur = f(Iterator.this.next); next }
    else throw new Error("next on empty iterator")
}
```

Closely related to map is the foreach method, which applies a given function to all elements of an iterator, but does not construct a list of results

```
def foreach(f: a => unit): unit =
  while (hasNext) { f(next) }
```

Filter. Method `filter` constructs an iterator which returns all elements of the original iterator that satisfy a criterion `p`.

```
  def filter(p: a => boolean) = new BufferedIterator[a] {
    private val source =
      Iterator.this.buffered
    private def skip
      { while (source.hasNext && !p(source.head)) { source.next } }
    def hasNext: boolean =
      { skip; source.hasNext }
    def next: a =
      { skip; source.next }
    def head: a =
      { skip; source.head; }
  }
```

In fact, `filter` returns instances of a subclass of iterators which are "buffered". A `BufferedIterator` object is an iterator which has in addition a method `head`. This method returns the element which would otherwise have been returned by `head`, but does not advance beyond that element. Hence, the element returned by `head` is returned again by the next call to `head` or `next`. Here is the definition of the `BufferedIterator` trait.

```
trait BufferedIterator[+a] extends Iterator[a] {
  def head: a
}
```

Since `map`, `flatMap`, `filter`, and `foreach` exist for iterators, it follows that for-comprehensions and for-loops can also be used on iterators. For instance, the application which prints the squares of numbers between 1 and 100 could have equivalently been expressed as follows.

```
for (i <- Iterator.range(1, 100))
  System.out.println(i * i)
```

Zip. Method `zip` takes another iterator and returns an iterator consisting of pairs of corresponding elements returned by the two iterators.

```
  def zip[b](that: Iterator[b]) = new Iterator[{a, b}] {
    def hasNext = Iterator.this.hasNext && that.hasNext
    def next = {Iterator.this.next, that.next}
  }
}
```

12.2 Constructing Iterators

Concrete iterators need to provide implementations for the two abstract methods `next` and `hasNext` in class `Iterator`. The simplest iterator is `Iterator.empty` which always returns an empty sequence:

```
object Iterator {
  object empty extends Iterator[Nothing] {
    def hasNext = false
    def next: a = throw new Error("next on empty iterator")
  }
```

A more interesting iterator enumerates all elements of an array. This iterator is constructed by the `fromArray` method, which is also defined in the object `Iterator`

```
  def fromArray[a](xs: Array[a]) = new Iterator[a] {
    private var i = 0
    def hasNext: boolean =
      i < xs.length
    def next: a =
      if (i < xs.length) { val x = xs(i) ; i = i + 1 ; x }
      else throw new Error("next on empty iterator")
  }
```

Another iterator enumerates an integer interval. The `Iterator.range` function returns an iterator which traverses a given interval of integer values. It is defined as follows.

```
object Iterator {
  def range(start: int, end: int) = new Iterator[int] {
    private var current = start
    def hasNext = current < end
    def next = {
      val r = current
      if (current < end) current = current + 1
      else throw new Error("end of iterator")
      r
    }
  }
}
```

All iterators seen so far terminate eventually. It is also possible to define iterators that go on forever. For instance, the following iterator returns successive integers from some start value[1].

[1]Due to the finite representation of type *int*, numbers will wrap around at 2^{31}.

```
def from(start: int) = new Iterator[int] {
  private var last = start - 1
  def hasNext = true
  def next = { last = last + 1; last }
}
```

12.3 Using Iterators

Here are two more examples how iterators are used. First, to print all elements of an array `xs: Array[int]`, one can write:

```
    Iterator.fromArray(xs) foreach (x =>
      System.out.println(x))
```

Or, using a for-comprehension:

```
    for (x <- Iterator.fromArray(xs))
      System.out.println(x)
```

As a second example, consider the problem of finding the indices of all the elements in an array of `doubles` greater than some `limit`. The indices should be returned as an iterator. This is achieved by the following expression.

```
import Iterator._
fromArray(xs)
.zip(from(0))
.filter(case {x, i} => x > limit)
.map(case {x, i} => i)
```

Or, using a for-comprehension:

```
import Iterator._
for ((x, i) <- fromArray(xs) zip from(0); x > limit)
yield i
```

Chapter 13

Implicit Parameters and Conversions

Implicit parameters and conversions are powerful tools for custimizing existing libraries and for creating high-level abstractions. As an example, let's start with an abstract class of semi-groups that support an unspecified add operation.

```
abstract class SemiGroup[a] {
  def add(x: a, y: a): a
}
```

Here's a subclass Monoid of SemiGroup which adds a unit element.

```
abstract class Monoid[a] extends SemiGroup[a] {
  def unit: a
}
```

Here are two implementations of monoids:

```
object stringMonoid extends Monoid[String] {
  def add(x: String, y: String): String = x.concat(y)
  def unit: String = ""
}

object intMonoid extends Monoid[int] {
  def add(x: int, y: int): int = x + y
  def unit: int = 0
}
```

A sum method, which works over arbitrary monoids, can be written in plain Scala as follows.

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

This sum method can be called as follows:

```
  sum(List("a", "bc", "def"))(stringMonoid)
  sum(List(1, 2, 3))(intMonoid)
```

All this works, but it is not very nice. The problem is that the monoid implementations have to be passed into all code that uses them. We would sometimes wish that the system could figure out the correct arguments automatically, similar to what is done when type arguments are inferred. This is what implicit parameters provide.

Implicit Parameters: The Basics

In Scala 2 there is a new **implicit** keyword that can be used at the beginning of a parameter list. Syntax:

```
  ParamClauses ::= {'(' [Param {',' Param}] ')'}
                   ['(' implicit Param {',' Param} ')']
```

If the keyword is present, it makes all parameters in the list implicit. For instance, the following version of sum has m as an implicit parameter.

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

As can be seen from the example, it is possible to combine normal and implicit parameters. However, there may only be one implicit parameter list for a method or constructor, and it must come last.

implicit can also be used as a modifier for definitions and declarations. Examples:

```
  implicit object stringMonoid extends Monoid[String] {
    def add(x: String, y: String): String = x.concat(y)
    def unit: String = ""
  }
  implicit object intMonoid extends Monoid[int] {
    def add(x: int, y: int): int = x + y
    def unit: int = 0
  }
```

The principal idea behind implicit parameters is that arguments for them can be left out from a method call. If the arguments corresponding to an implicit parameter section are missing, they are inferred by the Scala compiler.

The actual arguments that are eligible to be passed to an implicit parameter are all identifiers X that can be accessed at the point of the method call without a prefix and that denote an implicit definition or parameter.

If there are several eligible arguments which match the implicit parameter's type, the Scala compiler will chose a most specific one, using the standard rules of static overloading resolution. For instance, assume the call

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

in a context where `stringMonoid` and `intMonoid` are visible. We know that the formal type parameter `a` of `sum` needs to be instantiated to `int`. The only eligible value which matches the implicit formal parameter type `Monoid[int]` is `intMonoid` so this object will be passed as implicit parameter.

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

Implicit Conversions

Say you have an expression E of type T which is expected to type S. T does not conform to S and is not convertible to S by some other predefined conversion. Then the Scala compiler will try to apply as last resort an implicit conversion $I(E)$. Here, I is an identifier denoting an implicit definition or parameter that is accessible without a prefix at the point of the conversion, that can be applied to arguments of type T and whose result type conforms to the expected type S.

Implicit conversions can also be applied in member selections. Given a selection $E.x$ where x is not a member of the type E, the Scala compiler will try to insert an implicit conversion $I(E).x$, so that x is a member of $I(E)$.

Here is an example of an implicit conversion function that converts integers into instances of class `scala.Ordered`:

```
implicit def int2ordered(x: int): Ordered[int] = new Ordered[int] {
  def compare(y: int): int =
    if (x < y1) -1
    else if (x > y1) 1
    else 0
}
```

View Bounds

View bounds are convenient syntactic sugar for implicit parameters. Consider for instance a generic sort method:

```
def sort[a <% Ordered[a]](xs: List[a]): List[a] =
  if (xs.isEmpty || xs.tail.isEmpty) xs
```

```
    else {
      val {ys, zs} = xs.splitAt(xs.length / 2)
      merge(ys, zs)
    }
```

The view bounded type parameter `[a <% Ordered[a]]` expresses that `sort` is applicable to lists of type `a` such that there exists an implicit conversion from `a` to `Ordered[a]`. The definition is treated as a shorthand for the following method signature with an implicit parameter:

```
  def sort[a](xs: List[a])(implicit c: a => Ordered[a]): List[a] = ...
```

(Here, the parameter name *c* is chosen arbitrarily in a way that does not collide with other names in the program.)

As a more detailed example, consider the `merge` method that comes with the `sort` method above:

```
  def merge[a <% Ordered[a]](xs: List[a], ys: List[a]): List[a] =
    if (xs.isEmpty) ys
    else if (ys.isEmpty) xs
    else if (xs.head < ys.head) xs.head :: merge(xs.tail, ys)
    else if ys.head :: merge(xs, ys.tail)
```

After expanding view bounds and inserting implicit conversions, this method implementation becomes:

```
  def merge[a](xs: List[a], ys: List[a])
              (implicit c: a => Ordered[a]): List[a] =
    if (xs.isEmpty) ys
    else if (ys.isEmpty) xs
    else if (c(xs.head) < ys.head) xs.head :: merge(xs.tail, ys)
    else if ys.head :: merge(xs, ys.tail)(c)
```

The last two lines of this method definition illustrate two different uses of the implicit parameter *c*. It is applied in a conversion in the condition of the second to last line, and it is passed as implicit argument in the recursive call to `merge` on the last line.

Chapter 14

Combinator Parsing

In this chapter we describe how to write combinator parsers in Scala. Such parsers are constructed from predefined higher-order functions, so called *parser combinators*, that closely model the constructions of an EBNF grammar [Wir77].

As running example, we consider parsers for possibly nested lists of identifiers and numbers, which are described by the following context-free grammar.

letter	::=	/* all letters */
digit	::=	/* all digits */
ident	::=	letter {letter | digit }
number	::=	digit {digit}
list	::=	'(' [listElems] ')'
listElems	::=	expr [',' listElems]
expr	::=	ident | number | list

14.1 Simple Combinator Parsing

In this section we will only be concerned with the task of recognizing input strings, not with processing them. So we can describe parsers by the sets of input strings they accept. There are two fundamental operators over parsers: `&` expresses the sequential composition of a parser with another, while `|` expresses an alternative. These operations will both be defined as methods of a `Parser` class. We will also define constructors for the following primitive parsers:

`empty`	The parser that accepts the empty string
`failure(msg: String)`	The parser that accepts no string (`msg` stands for an error me
`chr(c: char)`	The parser that accepts the single-character string “c”.
`chrSuchThat(p: char => boolean)`	The parser that accepts single-character strings “c” for which $p(c)$ is true.

There are also the two higher-order parser combinators `opt`, expressing optionality and `rep`, expressing repetition. For any parser p, `opt(`p`)` yields a parser that accepts the strings accepted by p or else the empty string, while `rep(`p`)` accepts arbitrary sequences of the strings accepted by p. In EBNF, `opt(`p`)` corresponds to $[p]$ and `rep(`p`)` corresponds to $\{p\}$.

The central idea of parser combinators is that parsers can be produced by a straightforward rewrite of the grammar, replacing `::=` with `=`, sequencing with `&`, repetition `{...}` with `rep(...)` and optional occurrence `[...]` with `opt(...)`. Applying this process to the grammar of lists yields the following trait.

```
trait class ListParsers extends Parsers {
  def chrSuchThat(p: char => boolean): Parser
  def chr(c: char): Parser = chrSuchThat(d ==)

  def letter    : Parser = chr(Character.isLetter)
  def digit     : Parser = chr(Character.isDigit)

  def ident     : Parser = letter &&& rep(letter ||| digit)
  def number    : Parser = digit &&& rep(digit)
  def list      : Parser = chr('(') &&& opt(listElems) &&& chr(')')
  def listElems : Parser = expr &&& (chr(',') &&& listElems ||| empty)
  def expr      : Parser = ident ||| number ||| list
}
```

This class isolates the grammar from other aspects of parsing. It abstracts over the type of input and over the method used to parse a single character (represented by the abstract method `chr(p: char => boolean))`. The missing bits of information need to be supplied by code applying the parser class.

It remains to explain how to implement a library with the combinators described above. We will pack combinators and their underlying implementation in a base class `Parsers`, which is inherited by `ListParsers`. The first question to decide is which underlying representation type to use for a parser. We treat parsers here essentially as functions that take a datum of the input type `intype` and that yield a parse result of type `Option[intype]`. The `Option` type is predefined as follows.

```
abstract class Option[+a]
case object None extends Option[Nothing]
case class Some[a](x: a) extends Option[a]
```

A parser applied to some input either succeeds or fails. If it fails, it returns the constant None. If it succeeds, it returns a value of the form Some(in1) where in1 represents the input that remains to be parsed.

```
trait Parsers {
  type intype
  abstract class Parser {
    type Result = Option[intype]
    def apply(in: intype): Result
```

A parser also implements the combinators for sequence and alternative:

```
    /*** p &&& q applies first p, and if that succeeds, then q
     */
    def &&& (q: => Parser) = new Parser {
      def apply(in: intype): Result = Parser.this.apply(in) match {
        case None => None
        case Some(in1)  => q(in1)
      }
    }

    /*** p ||| q applies first p, and, if that fails, then q.
     */
    def ||| (q: => Parser) = new Parser {
      def apply(in: intype): Result = Parser.this.apply(in) match {
        case None => q(in)
        case s => s
      }
    }
```

The implementations of the primitive parsers empty and fail are trivial:

```
    val empty = new Parser { def apply(in: intype): Result = Some(in) }
    val fail  = new Parser { def apply(in: intype): Result = None }
```

The higher-order parser combinators opt and rep can be defined in terms of the combinators for sequence and alternative:

```
    def opt(p: Parser): Parser = p ||| empty;    // p? = (p | <empty>)
    def rep(p: Parser): Parser = opt(rep1(p));   // p* = [p+]
    def rep1(p: Parser): Parser = p &&& rep(p);  // p+ = p p*
  } // end Parser
```

To run combinator parsers, we still need to decide on a way to handle parser input. Several possibilities exist: The input could be represented as a list, as an array, or as a random access file. Note that the presented combinator parsers use backtracking to change from one alternative to another. Therefore, it must be possible to reset input to a point that was previously parsed. If one restricted the focus to

LL(1) grammars, a non-backtracking implementation of the parser combinators in class Parsers would also be possible. In that case sequential input methods based on (say) iterators or sequential files would also be possible.

In our example, we represent the input by a pair of a string, which contains the input phrase as a whole, and an index, which represents the portion of the input which has not yet been parsed. Since the input string does not change, just the index needs to be passed around as a result of individual parse steps. This leads to the following class of parsers that read strings:

```
class ParseString(s: String) extends Parsers {
  type intype = int
  def chrSuchThat(p: char => boolean) = new Parser {
    def apply(in: int): Parser#Result =
      if (in < s.length() && p(s charAt in)) Some(in + 1)
      else None
  }
  val input = 0
}
```

This class implements a method `chr(p: char => boolean)` and a value `input`. The `chr` method builds a parser that either reads a single character satisfying the given predicate `p` or fails. All other parsers over strings are ultimately implemented in terms of that method. The `input` value represents the input as a whole. In out case, it is simply value `0`, the start index of the string to be read.

Note `apply`'s result type, `Parser#Result`. This syntax selects the type element `Result` of the type `Parser`. It thus corresponds roughly to selecting a static inner class from some outer class in Java. Note that we could *not* have written `Parser.Result`, as the latter would express selection of the `Result` element from a *value* named `Parser`.

We have now extended the root class `Parsers` in two different directions: Class `ListParsers` defines a grammar of phrases to be parsed, whereas class `ParseString` defines a method by which such phrases are input. To write a concrete parsing application, we need to define both grammar and input method. We do this by combining two extensions of `Parsers` using a *mixin composition*. Here is the start of a sample application:

```
object Test {
  def main(args: Array[String]) {
    val ps = new ParseString(args(0)) with ListParsers
  }
```

The last line above creates a new family of parsers by composing class `ListParsers` with class `ParseString`. The two classes share the common superclass `Parsers`. The abstract method `chr` in `ListParsers` is implemented by class `ParseString`.

To run the parser, we apply the start symbol of the grammar `expr` the argument codeinput and observe the result:

```
      ps.expr(ps.input) match {
        case Some(n) =>
          System.out.println("parsed: " + args(0).substring(0, n));
        case None =>
          System.out.println("nothing parsed");
      }
    }
  }// end Test
```

Note the syntax `ps.expr(input)`, which treats the `expr` parser as if it was a function. In Scala, objects with `apply` methods can be applied directly to arguments as if they were functions.

Here is an example run of the program above:

```
> java examples.Test "(x,1,(y,z))"
parsed: (x,1,(y,z))
> java examples.Test "(x,,1,(y,z))"
nothing parsed
```

14.2 Parsers that Produce Results

The combinator library of the previous section does not support the generation of output from parsing. But usually one does not just want to check whether a given string belongs to the defined language, one also wants to convert the input string into some internal representation such as an abstract syntax tree.

In this section, we modify our parser library to build parsers that produce results. We will make use of the for-comprehensions introduced in Chapter 9. The basic combinator of sequential composition, formerly `p &&& q`, now becomes

```
for (x <- p; y <- q) yield e .
```

Here, the names `x` and `y` are bound to the results of executing the parsers `p` and `q`. `e` is an expression that uses these results to build the tree returned by the composed parser.

Before describing the implementation of the new parser combinators, we explain how the new building blocks are used. Say we want to modify our list parser so that it returns an abstract syntax tree of the parsed expression. Syntax trees are given by the following class hierarchy:

```
abstract class Tree
case class Id (s: String)          extends Tree
case class Num(n: int)           extends Tree
case class Lst(elems: List[Tree]) extends Tree
```

That is, a syntax tree is an identifier, an integer number, or a `Lst` node with a list of trees as descendants.

As a first step towards parsers that produce results we define three little parsers that return a single read character as result.

```
trait CharParsers extends Parsers {
  def any: Parser[char]
  def chr(ch: char): Parser[char] =
    for (c <- any if c == ch) yield c
  def chrSuchThat(p: char => boolean): Parser[char] =
    for (c <- any if p(c)) yield c
}
```

The `any` parser succeeds with the first character of remaining input as long as input is nonempty. It is abstract in class `ListParsers` since we want to abstract in this class from the concrete input method used. The two `chr` parsers return as before the first input character if it equals a given character or matches a given predicate. They are now implemented in terms of `any`.

The next level is represented by parsers reading identifiers, numbers and lists. Here is a parser for identifiers.

```
trait ListParsers extends CharParsers {
  def ident: Parser[Tree] =
    for {
      c: char <- chrSuchThat(Character.isLetter)
      cs: List[char] <- rep(chrSuchThat(Character.isLetterOrDigit))
    } yield Id((c :: cs).mkString("", "", ""))
```

Remark: Because `chrSuchThat(...)` returns a single character, its repetition `rep(chrSuchThat(...))` returns a list of characters. The **`yield`** part of the for-comprehension converts all intermediate results into an `Id` node with a string as element. To convert the read characters into a string, it conses them into a single list, and invokes the `mkString` method on the result.

Here is a parser for numbers:

```
  def number: Parser[Tree] =
    for {
      d: char <- chrSuchThat(Character.isDigit)
      ds: List[char] <- rep(chrSuchThat(Character.isDigit))
    } yield Num(((d - '0') /: ds) ((x, digit) => x * 10 + digit - '0'))
```

Intermediate results are in this case the leading digit of the read number, followed by a list of remaining digits. The **yield** part of the for-comprehension reduces these to a number by a fold-left operation.

Here is a parser for lists:

```
    def list: Parser[Tree] =
      for {
        _ <- chr('(')
        es <- listElems ||| succeed(List())
        _ <- chr(')')
      } yield Lst(es)

    def listElems: Parser[List[Tree]] =
      for {
        x <- expr
        xs <- chr(',') &&& listElems ||| succeed(List())
      } yield x :: xs
```

The `list` parser returns a `Lst` node with a list of trees as elements. That list is either the result of `listElems`, or, if that fails, the empty list (expressed here as: the result of a parser which always succeeds with the empty list as result).

The highest level of our grammar is represented by function `expr`:

```
    def expr: Parser[Tree] =
      ident ||| number ||| list
  }// end ListParsers.
```

We now present the parser combinators that support the new scheme. Parsers that succeed now return a parse result besides the un-consumed input.

```
  trait Parsers {
    type intype
    abstract class Parser[a] {
      type Result = Option[{a, intype}]
      def apply(in: intype): Result
```

Parsers are parameterized with the type of their result. The class `Parser[a]` now defines new methods `map`, `flatMap` and `filter`. The **for** expressions are mapped by the compiler to calls of these functions using the scheme described in Chapter 9. For parsers, these methods are implemented as follows.

```
      def filter(pred: a => boolean) = new Parser[a] {
        def apply(in: intype): Result = Parser.this.apply(in) match {
          case None => None
          case Some{x, in1} => if (pred(x)) Some{x, in1} else None
        }
      }
```

```
    def map[b](f: a => b) = new Parser[b] {
      def apply(in: intype): Result = Parser.this.apply(in) match {
        case None => None
        case Some{x, in1} => Some{f(x), in1}
      }
    }
    def flatMap[b](f: a => Parser[b]) = new Parser[b] {
      def apply(in: intype): Result = Parser.this.apply(in) match {
        case None => None
        case Some{x, in1} => f(x).apply(in1)
      }
    }
```

The `filter` method takes as parameter a predicate *p* which it applies to the results of the current parser. If the predicate is false, the parser fails by returning `None`; otherwise it returns the result of the current parser. The `map` method takes as parameter a function *f* which it applies to the results of the current parser. The `flatMap` takes as parameter a function `f` which returns a parser. It applies `f` to the result of the current parser and then continues with the resulting parser. The `|||` method is essentially defined as before. The `&&&` method can now be defined in terms of **`for`**.

```
    def ||| (p: => Parser[a]) = new Parser[a] {
      def apply(in: intype): Result = Parser.this.apply(in) match {
        case None => p(in)
        case s => s
      }
    }

    def &&& [b](p: => Parser[b]): Parser[b] =
      for (_ <- this; x <- p) yield x
  }// end Parser
```

The primitive parser `succeed` replaces `empty`. It consumes no input and returns its parameter as result.

```
  def succeed[a](x: a) = new Parser[a] {
    def apply(in: intype) = Some{x, in}
  }
```

The parser combinators `rep` and `opt` now also return results. `rep` returns a list which contains as elements the results of each iteration of its sub-parser. `opt` returns a list which is either empty or returns as single element the result of the optional parser.

```
  def rep[a](p: Parser[a]): Parser[List[a]] =
    rep1(p) ||| succeed(List())

  def rep1[a](p: Parser[a]): Parser[List[a]] =
      for (x <- p; xs <- rep(p)) yield x :: xs

    def opt[a](p: Parser[a]): Parser[List[a]] =
      (for (x <- p) yield List(x)) ||| succeed(List())
  } // end Parsers
```

The root class Parsers abstracts over which kind of input is parsed. As before, we determine the input method by a separate class. Here is ParseString, this time adapted to parsers that return results. It defines now the method any, which returns the first input character.

```
  class ParseString(s: String) extends Parsers {
    type intype = int
    val input = 0
    def any = new Parser[char] {
      def apply(in: int): Parser[char]#Result =
        if (in < s.length()) Some{s charAt in, in + 1} else None
    }
  }
```

The rest of the application is as before. Here is a test program which constructs a list parser over strings and prints out the result of applying it to the command line argument.

```
  object Test {
    def main(args: Array[String]) {
      val ps = new ParseString(args(0)) with ListParsers
      ps.expr(ps.input) match {
        case Some{list, _} => System.out.println("parsed: " + list)
        case None => System.out.println("nothing parsed")
      }
    }
  }
```

Exercise 14.2.1 The parsers we have defined so far can succeed even if there is some input beyond the parsed text. To prevent this, one needs a parser which recognizes the end of input. Redesign the parser library so that such a parser can be introduced. Which classes need to be modified?

Chapter 15

Hindley/Milner Type Inference

This chapter demonstrates Scala's data types and pattern matching by developing a type inference system in the Hindley/Milner style [Mil78]. The source language for the type inferencer is lambda calculus with a let construct called Mini-ML. Abstract syntax trees for the Mini-ML are represented by the following data type of `Terms`.

```
abstract class Term {}
case class Var(x: String) extends Term {
  override def toString = x
}
case class Lam(x: String, e: Term) extends Term {
  override def toString = "(\\" + x + "." + e + ")"
}
case class App(f: Term, e: Term) extends Term {
  override def toString = "(" + f + " " + e + ")"
}
case class Let(x: String, e: Term, f: Term) extends Term {
  override def toString = "let " + x + " = " + e + " in " + f
}
```

There are four tree constructors: `Var` for variables, `Lam` for function abstractions, `App` for function applications, and `Let` for let expressions. Each case class overrides the `toString` method of class `Any`, so that terms can be printed in legible form.

We next define the types that are computed by the inference system.

```
sealed abstract class Type {}
case class Tyvar(a: String) extends Type {
  override def toString = a
}
case class Arrow(t1: Type, t2: Type) extends Type {
  override def toString = "(" + t1 + "->" + t2 + ")"
}
```

```
case class Tycon(k: String, ts: List[Type]) extends Type {
  override def toString =
    k + (if (ts.isEmpty) "" else ts.mkString("[", ",", "]"))
}
```

There are three type constructors: `Tyvar` for type variables, `Arrow` for function types and `Tycon` for type constructors such as `boolean` or `List`. Type constructors have as component a list of their type parameters. This list is empty for type constants such as `boolean`. Again, the type constructors implement the `toString` method in order to display types legibly.

Note that `Type` is a **sealed** class. This means that no subclasses or data constructors that extend `Type` can be formed outside the sequence of definitions in which `Type` is defined. This makes `Type` a *closed* algebraic data type with exactly three alternatives. By contrast, type `Term` is an *open* algebraic type for which further alternatives can be defined.

The main parts of the type inferencer are contained in object `typeInfer`. We start with a utility function which creates fresh type variables:

```
object typeInfer {
  private var n: Int = 0
  def newTyvar(): Type = { n = n + 1 ; Tyvar("a" + n) }
```

We next define a class for substitutions. A substitution is an idempotent function from type variables to types. It maps a finite number of type variables to some types, and leaves all other type variables unchanged. The meaning of a substitution is extended point-wise to a mapping from types to types.

```
  abstract class Subst extends Function1[Type,Type] {

    def lookup(x: Tyvar): Type

    def apply(t: Type): Type = t match {
      case tv @ Tyvar(a) => val u = lookup(tv); if (t == u) t else apply(u);
      case Arrow(t1, t2) => Arrow(apply(t1), apply(t2))
      case Tycon(k, ts) => Tycon(k, ts map apply)
    }

    def extend(x: Tyvar, t: Type) = new Subst {
      def lookup(y: Tyvar): Type = if (x == y) t else Subst.this.lookup(y)
    }
  }
  val emptySubst = new Subst { def lookup(t: Tyvar): Type = t }
```

We represent substitutions as functions, of type `Type => Type`. This is achieved by making class `Subst` inherit from the unary function type `Function1[Type, Type]`[1]. To be an instance of this type, a substitution `s` has to implement an `apply` method that takes a `Type` as argument and yields another `Type` as result. A function application `s(t)` is then interpreted as `s.apply(t)`.

The `lookup` method is abstract in class `Subst`. There are two concrete forms of substitutions which differ in how they implement this method. One form is defined by the `emptySubst` value, the other is defined by the `extend` method in class `Subst`.

The next data type describes type schemes, which consist of a type and a list of names of type variables which appear universally quantified in the type scheme. For instance, the type scheme $\forall a \forall b. a \rightarrow b$ would be represented in the type checker as:

```
TypeScheme(List(Tyvar("a"), Tyvar("b")), Arrow(Tyvar("a"), Tyvar("b"))) .
```

The class definition of type schemes does not carry an extends clause; this means that type schemes extend directly class `AnyRef`. Even though there is only one possible way to construct a type scheme, a case class representation was chosen since it offers convenient ways to decompose an instance of this type into its parts.

```
case class TypeScheme(tyvars: List[Tyvar], tpe: Type) {
  def newInstance: Type = {
    (emptySubst /: tyvars) ((s, tv) => s.extend(tv, newTyvar())) (tpe)
  }
}
```

Type scheme objects come with a method `newInstance`, which returns the type contained in the scheme after all universally type variables have been renamed to fresh variables. The implementation of this method folds (with `/:`) the type scheme's type variables with an operation which extends a given substitution `s` by renaming a given type variable `tv` to a fresh type variable. The resulting substitution renames all type variables of the scheme to fresh ones. This substitution is then applied to the type part of the type scheme.

The last type we need in the type inferencer is `Env`, a type for environments, which associate variable names with type schemes. They are represented by a type alias `Env` in module `typeInfer`:

```
type Env = List[{String, TypeScheme}]
```

There are two operations on environments. The `lookup` function returns the type scheme associated with a given name, or **`null`** if the name is not recorded in the environment.

[1] The class inherits the function type as a mixin rather than as a direct superclass. This is because in the current Scala implementation, the `Function1` type is a Java interface, which cannot be used as a direct superclass of some other class.

```
def lookup(env: Env, x: String): TypeScheme = env match {
  case List() => null
  case {y, t} :: env1 => if (x == y) t else lookup(env1, x)
}
```

The gen function turns a given type into a type scheme, quantifying over all type variables that are free in the type, but not in the environment.

```
def gen(env: Env, t: Type): TypeScheme =
  TypeScheme(tyvars(t) diff tyvars(env), t)
```

The set of free type variables of a type is simply the set of all type variables which occur in the type. It is represented here as a list of type variables, which is constructed as follows.

```
def tyvars(t: Type): List[Tyvar] = t match {
  case tv @ Tyvar(a) =>
    List(tv)
  case Arrow(t1, t2) =>
    tyvars(t1) union tyvars(t2)
  case Tycon(k, ts) =>
    (List[Tyvar]() /: ts) ((tvs, t) => tvs union tyvars(t))
}
```

Note that the syntax `tv @ ...` in the first pattern introduces a variable which is bound to the pattern that follows. Note also that the explicit type parameter `[Tyvar]` in the expression of the third clause is needed to make local type inference work.

The set of free type variables of a type scheme is the set of free type variables of its type component, excluding any quantified type variables:

```
def tyvars(ts: TypeScheme): List[Tyvar] =
  tyvars(ts.tpe) diff ts.tyvars
```

Finally, the set of free type variables of an environment is the union of the free type variables of all type schemes recorded in it.

```
def tyvars(env: Env): List[Tyvar] =
  (List[Tyvar]() /: env) ((tvs, nt) => tvs union tyvars(nt._2))
```

A central operation of Hindley/Milner type checking is unification, which computes a substitution to make two given types equal (such a substitution is called a *unifier*). Function mgu computes the most general unifier of two given types t and u under a pre-existing substitution s. That is, it returns the most general substitution s' which extends s, and which makes $s'(t)$ and $s'(u)$ equal types.

```
def mgu(t: Type, u: Type, s: Subst): Subst = {s(t), s(u)} match {
  case {Tyvar(a), Tyvar(b)} if (a == b) =>
```

```
      s
    case {Tyvar(a), _} if !(tyvars(u) contains a) =>
      s.extend(Tyvar(a), u)
    case {_, Tyvar(a)} =>
      mgu(u, t, s)
    case {Arrow(t1, t2), Arrow(u1, u2)} =>
      mgu(t1, u1, mgu(t2, u2, s))
    case {Tycon(k1, ts), Tycon(k2, us)} if (k1 == k2) =>
      (s /: (ts zip us)) ((s, tu) => mgu(tu._1, tu._2, s))
    case _ =>
      throw new TypeError("cannot unify " + s(t) + " with " + s(u))
  }
```

The mgu function throws a TypeError exception if no unifier substitution exists. This can happen because the two types have different type constructors at corresponding places, or because a type variable is unified with a type that contains the type variable itself. Such exceptions are modeled here as instances of case classes that inherit from the predefined Exception class.

```
case class TypeError(s: String) extends Exception(s) {}
```

The main task of the type checker is implemented by function tp. This function takes as parameters an environment *env*, a term *e*, a proto-type *t*, and a pre-existing substitution *s*. The function yields a substitution s' that extends *s* and that turns $s'(env) \vdash e : s'(t)$ into a derivable type judgment according to the derivation rules of the Hindley/Milner type system [Mil78]. A TypeError exception is thrown if no such substitution exists.

```
def tp(env: Env, e: Term, t: Type, s: Subst): Subst = {
  current = e
  e match {
    case Var(x) =>
      val u = lookup(env, x)
      if (u == null) throw new TypeError("undefined: " + x)
      else mgu(u.newInstance, t, s)

    case Lam(x, e1) =>
      val a, b = newTyvar()
      val s1 = mgu(t, Arrow(a, b), s)
      val env1 = {x, TypeScheme(List(), a)} :: env
      tp(env1, e1, b, s1)

    case App(e1, e2) =>
      val a = newTyvar()
      val s1 = tp(env, e1, Arrow(a, t), s)
      tp(env, e2, a, s1)
      case Let(x, e1, e2) =>
        val a = newTyvar()
        val s1 = tp(env, e1, a, s)
        tp({x, gen(env, s1(a))} :: env, e2, t, s1)
    }
  }
  var current: Term = null
```

To aid error diagnostics, the `tp` function stores the currently analyzed sub-term in variable `current`. Thus, if type checking is aborted with a `TypeError` exception, this variable will contain the subterm that caused the problem.

The last function of the type inference module, `typeOf`, is a simplified facade for `tp`. It computes the type of a given term e in a given environment env. It does so by creating a fresh type variable a, computing a typing substitution that makes $env \vdash e : a$ into a derivable type judgment, and returning the result of applying the substitution to a.

```
  def typeOf(env: Env, e: Term): Type = {
    val a = newTyvar()
    tp(env, e, a, emptySubst)(a)
  }
}// end typeInfer
```

To apply the type inferencer, it is convenient to have a predefined environment that contains bindings for commonly used constants. The module `predefined` defines an environment `env` that contains bindings for the types of booleans, numbers and lists together with some primitive operations over them. It also defines a fixed point operator `fix`, which can be used to represent recursion.

```
object predefined {
  val booleanType = Tycon("Boolean", List())
  val intType = Tycon("Int", List())
  def listType(t: Type) = Tycon("List", List(t))

  private def gen(t: Type): typeInfer.TypeScheme = typeInfer.gen(List(), t)
  private val a = typeInfer.newTyvar()
  val env = List(
    {"true", gen(booleanType)},
    {"false", gen(booleanType)},
    {"if", gen(Arrow(booleanType, Arrow(a, Arrow(a, a))))},
    {"zero", gen(intType)},
    {"succ", gen(Arrow(intType, intType))},
    {"nil", gen(listType(a))},
    {"cons", gen(Arrow(a, Arrow(listType(a), listType(a))))},
    {"isEmpty", gen(Arrow(listType(a), booleanType))},
      {"head", gen(Arrow(listType(a), a))},
      {"tail", gen(Arrow(listType(a), listType(a)))},
      {"fix", gen(Arrow(Arrow(a, a), a))}
    )
  }
```

Here's an example how the type inferencer can be used. Let's define a function showType which returns the type of a given term computed in the predefined environment Predefined.env:

```
object testInfer {
  def showType(e: Term): String =
    try {
      typeInfer.typeOf(predefined.env, e).toString
    } catch {
      case typeInfer.TypeError(msg) =>
        "\n cannot type: " + typeInfer.current +
        "\n reason: " + msg
    }
```

Then the application

```
> testInfer.showType(Lam("x", App(App(Var("cons"), Var("x")), Var("nil"))))
```

would give the response

```
> (a6->List[a6])
```

To make the type inferencer more useful, we complete it with a parser. Function main of module testInfer parses and typechecks a Mini-ML expression which is given as the first command line argument.

```
  def main(args: Array[String]) {
    val ps = new ParseString(args(0)) with MiniMLParsers
    ps.all(ps.input) match {
      case Some{term, _} =>
        System.out.println("" + term + ": " + showType(term))
      case None =>
        System.out.println("syntax error")
    }
  }
}// testInfer
```

To do the parsing, method main uses the combinator parser scheme of Chapter 14. It creates a parser family ps as a mixin composition of parsers that understand MiniML (but do not know where input comes from) and parsers that read input from a given string. The MiniMLParsers object implements parsers for the following grammar.

```
term  ::= "\" ident "." term
        |  term1 {term1}
        |  "let" ident "=" term "in" term
term1 ::= ident
        |  "(" term ")"
all   ::= term ";"
```

Input as a whole is described by the production `all`; it consists of a term followed by a semicolon. We allow "whitespace" consisting of one or more space, tabulator or newline characters between any two lexemes (this is not reflected in the grammar above). Identifiers are defined as in Chapter 14 except that an identifier cannot be one of the two reserved words "let" and "in".

```
trait MiniMLParsers extends CharParsers {

  /** whitespace */
  def whitespace = rep{chr(' ') ||| chr('\t') ||| chr('\n')}

  /** A given character, possible preceded by whitespace */
  def wschr(ch: char) = whitespace &&& chr(ch)

  /** identifiers or keywords */
  def id: Parser[String] =
    for {
      c: char <- whitespace &&& chrSuchThat(Character.isLetter);
      cs: List[char] <- rep(chrSuchThat(Character.isLetterOrDigit))
    } yield (c :: cs).mkString("", "", "")

  /** Non-keyword identifiers */
  def ident: Parser[String] =
    for { s <- id if s != "let" && s != "in" } yield s

  /** term = '\' ident '.' term | term1 {term1} | let ident "=" term in term */
  def term: Parser[Term] = (
    ( for {
        _ <- wschr('\\')
        x <- ident
        _ <- wschr('.')
        t <- term
      } yield Lam(x, t): Term )
    |||
    ( for {
        letid <- id if letid == "let"
        x <- ident
        _ <- wschr('=')
        t <- term;
        inid <- id; inid == "in"
        c <- term
      } yield Let(x, t, c) )
    |||
    ( for {
        t <- term1
        ts <- rep(term1)
      } yield (t /: ts)((f, arg) => App(f, arg)) )
  )

  /** term1 = ident | '(' term ')' */
  def term1: Parser[Term] = (
    ( for { s <- ident }
      yield Var(s): Term )
    |||
    ( for {
        _ <- wschr('(')
        t <- term
        _ <- wschr(')')
      } yield t )
  )

  /** all = term ';' */
  def all: Parser[Term] =
    for {
      t <- term
      _ <- wschr(';')
    } yield t
}
```

Here are some sample MiniML programs and the output the type inferencer gives for each of them:

```
> java testInfer
| "\x.\f.f(f x);"
(\x.(\f.(f (f x)))): (a8->((a8->a8)->a8))

> java testInfer
| "let id = \x.x
|  in if (id true) (id nil) (id (cons zero nil));"
let id = (\x.x) in (((if (id true)) (id nil)) (id ((cons zero) nil))): List[Int]

> java testInfer
| "let id = \x.x
|  in if (id true) (id nil);"
```

```
let id = (\x.x) in ((if (id true)) (id nil)): (List[a13]->List[a13])

> java testInfer
| "let length = fix (\len.\xs.
|    if (isEmpty xs)
|      zero
|      (succ (len (tail xs))))
|  in (length nil);"
let length = (fix (\len.(\xs.(((if (isEmpty xs)) zero)
(succ (len (tail xs))))))) in (length nil): Int

> java testInfer
| "let id = \x.x
|  in if (id true) (id nil) zero;"
let id = (\x.x) in (((if (id true)) (id nil)) zero):
 cannot type: zero
 reason: cannot unify Int with List[a14]
```

Exercise 15.0.2 Using the parser library constructed in Exercise Exercise 14.2.1, modify the MiniML parser library so that no marker ";" is necessary for indicating the end of input.

Exercise 15.0.3 Extend the Mini-ML parser and type inferencer with a `letrec` construct which allows the definition of recursive functions. Syntax:

```
letrec ident "=" term in term .
```

The typing of `letrec` is as for `let`, except that the defined identifier is visible in the defining expression. Using `letrec`, the `length` function for lists can now be defined as follows.

```
letrec length = \xs.
  if (isEmpty xs)
    zero
    (succ (length (tail xs)))
in ...
```

Chapter 16

Abstractions for Concurrency

This section reviews common concurrent programming patterns and shows how they can be implemented in Scala.

16.1 Signals and Monitors

Example 16.1.1 The *monitor* provides the basic means for mutual exclusion of processes in Scala. Every instance of class AnyRef can be used as a monitor by calling one or more of the methods below.

```
def synchronized[a] (e: => a): a
def wait()
def wait(msec: long)
def notify()
def notifyAll()
```

The synchronized method executes its argument computation e in mutual exclusive mode – at any one time, only one thread can execute a synchronized argument of a given monitor.

Threads can suspend inside a monitor by waiting on a signal. Threads that call the wait method wait until a notify method of the same object is called subsequently by some other thread. Calls to notify with no threads waiting for the signal are ignored.

There is also a timed form of wait, which blocks only as long as no signal was received or the specified amount of time (given in milliseconds) has elapsed. Furthermore, there is a notifyAll method which unblocks all threads which wait for the signal. These methods, as well as class Monitor are primitive in Scala; they are implemented in terms of the underlying runtime system.

Typically, a thread waits for some condition to be established. If the condition does not hold at the time of the wait call, the thread blocks until some other thread has established the condition. It is the responsibility of this other thread to wake up waiting processes by issuing a `notify` or `notifyAll`. Note however, that there is no guarantee that a waiting process gets to run immediately after the call to notify is issued. It could be that other processes get to run first which invalidate the condition again. Therefore, the correct form of waiting for a condition *C* uses a while loop:

```
while (!C) wait()
```

As an example of how monitors are used, here is is an implementation of a bounded buffer class.

```
class BoundedBuffer[a](N: int) {
  var in = 0, out = 0, n = 0
  val elems = new Array[a](N)

  def put(x: a) = synchronized {
    while (n >= N) wait()
    elems(in) = x ; in = (in + 1) % N ; n = n + 1
    if (n == 1) notifyAll()
  }

  def get: a = synchronized {
    while (n == 0) wait()
    val x = elems(out) ; out = (out + 1) % N ; n = n - 1
    if (n == N - 1) notifyAll()
    x
  }
}
```

And here is a program using a bounded buffer to communicate between a producer and a consumer process.

```
import scala.concurrent.ops._
...
val buf = new BoundedBuffer[String](10)
spawn { while (true) { val s = produceString ; buf.put(s) } }
spawn { while (true) { val s = buf.get ; consumeString(s) } }
}
```

The spawn method spawns a new thread which executes the expression given in the parameter. It is defined in object `concurrent.ops` as follows.

```
def spawn(p: => unit) {
  val t = new Thread() { override def run() = p; }
  t.start()
}
```

16.2 SyncVars

A synchronized variable (or syncvar for short) offers `get` and `put` operations to read and set the variable. `get` operations block until the variable has been defined. An `unset` operation resets the variable to undefined state.

Here's the standard implementation of synchronized variables.

```
package scala.concurrent
class SyncVar[a] {
  private var isDefined: boolean = false
  private var value: a = _
  def get = synchronized {
    while (!isDefined) wait()
    value
  }
  def set(x: a) = synchronized {
    value = x ; isDefined = true ; notifyAll()
  }
  def isSet: boolean = synchronized {
    isDefined
  }
  def unset = synchronized {
    isDefined = false;
  }
}
```

16.3 Futures

A *future* is a value which is computed in parallel to some other client thread, to be used by the client thread at some future time. Futures are used in order to make good use of parallel processing resources. A typical usage is:

```
import scala.concurrent.ops._
...
val x = future(someLengthyComputation)
anotherLengthyComputation
val y = f(x()) + g(x())
```

The `future` method is defined in object `scala.concurrent.ops` as follows.

```
def future[a](p: => a): unit => a = {
    val result = new SyncVar[a]
    fork { result.set(p) }
    (() => result.get)
  }
```

The `future` method gets as parameter a computation `p` to be performed. The type of the computation is arbitrary; it is represented by `future`'s type parameter `a`. The `future` method defines a guard `result`, which takes a parameter representing the result of the computation. It then forks off a new thread that computes the result and invokes the `result` guard when it is finished. In parallel to this thread, the function returns an anonymous function of type `a`. When called, this functions waits on the result guard to be invoked, and, once this happens returns the result argument. At the same time, the function reinvokes the `result` guard with the same argument, so that future invocations of the function can return the result immediately.

16.4 Parallel Computations

The next example presents a function `par` which takes a pair of computations as parameters and which returns the results of the computations in another pair. The two computations are performed in parallel.

The function is defined in object `scala.concurrent.ops` as follows.

```
def par[a, b](xp: => a, yp: => b): {a, b} = {
  val y = new SyncVar[b]
  spawn { y set yp }
  {xp, y.get}
}
```

Defined in the same place is a function `replicate` which performs a number of replicates of a computation in parallel. Each replication instance is passed an integer number which identifies it.

```
def replicate(start: int, end: int)(p: int => unit) {
  if (start == end)
    {}
  else if (start + 1 == end)
    p(start)
  else {
    val mid = (start + end) / 2
    spawn { replicate(start, mid)(p) }
    replicate(mid, end)(p)
  }
}
```

The next function uses `replicate` to perform parallel computations on all elements of an array.

```
def parMap[a,b](f: a => b, xs: Array[a]): Array[b] = {
  val results = new Array[b](xs.length)
  replicate(0, xs.length) { i => results(i) = f(xs(i)) }
  results
}
```

16.5 Semaphores

A common mechanism for process synchronization is a *lock* (or: *semaphore*). A lock offers two atomic actions: *acquire* and *release*. Here's the implementation of a lock in Scala:

```
package scala.concurrent

class Lock {
  var available = true
  def acquire = synchronized {
    while (!available) wait()
    available = false
  }
  def release = synchronized {
    available = true
    notify()
  }
}
```

16.6 Readers/Writers

A more complex form of synchronization distinguishes between *readers* which access a common resource without modifying it and *writers* which can both access and modify it. To synchronize readers and writers we need to implement operations *startRead*, *startWrite*, *endRead*, *endWrite*, such that:

- there can be multiple concurrent readers,
- there can only be one writer at one time,
- pending write requests have priority over pending read requests, but don't preempt ongoing read operations.

The following implementation of a readers/writers lock is based on the *mailbox* concept (see Section 16.10).

```
import scala.concurrent._

class ReadersWriters {
  val m = new MailBox
  private case class Writers(n: int), Readers(n: int) { m send this; }
  Writers(0); Readers(0)
  def startRead = m receive {
    case Writers(n) if n == 0 => m receive {
      case Readers(n) => Writers(0) ; Readers(n+1)
    }
  }
  def startWrite = m receive {
    case Writers(n) =>
      Writers(n+1)
      m receive { case Readers(n) if n == 0 => }
  }
  def endRead = m receive {
    case Readers(n) => Readers(n-1)
  }
  def endWrite = m receive {
    case Writers(n) => Writers(n-1) ; if (n == 0) Readers(0)
  }
}
```

16.7 Asynchronous Channels

A fundamental way of interprocess communication is the asynchronous channel. Its implementation makes use the following simple class for linked lists:

```
class LinkedList[a] {
  var elem: a = _
  var next: LinkedList[a] = null
}
```

To facilitate insertion and deletion of elements into linked lists, every reference into a linked list points to the node which precedes the node which conceptually forms the top of the list. Empty linked lists start with a dummy node, whose successor is **null**.

The channel class uses a linked list to store data that has been sent but not read yet. At the opposite end, threads that wish to read from an empty channel, register their presence by incrementing the `nreaders` field and waiting to be notified.

```
package scala.concurrent
```

```
class Channel[a] {
  class LinkedList[a] {
    var elem: a = _
    var next: LinkedList[a] = null
  }
  private var written = new LinkedList[a]
  private var lastWritten = written
  private var nreaders = 0

  def write(x: a) = synchronized {
    lastWritten.elem = x
    lastWritten.next = new LinkedList[a]
    lastWritten = lastWritten.next
    if (nreaders > 0) notify()
  }

  def read: a = synchronized {
    if (written.next == null) {
      nreaders = nreaders + 1; wait(); nreaders = nreaders - 1
    }
    val x = written.elem
    written = written.next
    x
  }
}
```

16.8 Synchronous Channels

Here's an implementation of synchronous channels, where the sender of a message blocks until that message has been received. Synchronous channels only need a single variable to store messages in transit, but three signals are used to coordinate reader and writer processes.

```
package scala.concurrent

class SyncChannel[a] {
  private var data: a = _
  private var reading = false
  private var writing = false

  def write(x: a) = synchronized {
    while (writing) wait()
    data = x
    writing = true
```

```
      if (reading) notifyAll()
      else while (!reading) wait()
    }

    def read: a = synchronized {
      while (reading) wait()
      reading = true
      while (!writing) wait()
      val x = data
      writing = false
      reading = false
      notifyAll()
      x
    }
  }
```

16.9 Workers

Here's an implementation of a *compute server* in Scala. The server implements a `future` method which evaluates a given expression in parallel with its caller. Unlike the implementation in Section 16.3 the server computes futures only with a predefined number of threads. A possible implementation of the server could run each thread on a separate processor, and could hence avoid the overhead inherent in context-switching several threads on a single processor.

```
  import scala.concurrent._, scala.concurrent.ops._

  class ComputeServer(n: int) {

    private abstract class Job {
      type t
      def task: t
      def ret(x: t)
    }

    private val openJobs = new Channel[Job]()

    private def processor(i: int) {
      while (true) {
        val job = openJobs.read
        job.ret(job.task)
      }
    }
    def future[a](p: => a): () => a = {
      val reply = new SyncVar[a]()
      openJobs.write{
        new Job {
          type t = a
          def task = p
          def ret(x: a) = reply.set(x)
        }
      }
      () => reply.get
    }

    spawn(replicate(0, n) { processor })
  }
```

Expressions to be computed (i.e. arguments to calls of `future`) are written to the `openJobs` channel. A *job* is an object with

- An abstract type `t` which describes the result of the compute job.
- A parameterless `task` method of type `t` which denotes the expression to be computed.
- A **`return`** method which consumes the result once it is computed.

The compute server creates *n* `processor` processes as part of its initialization. Every such process repeatedly consumes an open job, evaluates the job's `task` method and passes the result on to the job's **`return`** method. The polymorphic `future` method creates a new job where the **`return`** method is implemented by a guard named `reply` and inserts this job into the set of open jobs by calling the `isOpen` guard. It then waits until the corresponding `reply` guard is called.

The example demonstrates the use of abstract types. The abstract type `t` keeps track of the result type of a job, which can vary between different jobs. Without abstract types it would be impossible to implement the same class to the user in a statically type-safe way, without relying on dynamic type tests and type casts.

Here is some code which uses the compute server to evaluate the expression `41 + 1`.

```
  object Test with Executable {
    val server = new ComputeServer(1)
    val f = server.future(41 + 1)
    Console.println(f())
  }
```

16.10 Mailboxes

Mailboxes are high-level, flexible constructs for process synchronization and communication. They allow sending and receiving of messages. A *message* in this context is an arbitrary object. There is a special message TIMEOUT which is used to signal a time-out.

```
case object TIMEOUT
```

Mailboxes implement the following signature.

```
class MailBox {
  def send(msg: Any)
  def receive[a](f: PartialFunction[Any, a]): a
  def receiveWithin[a](msec: long)(f: PartialFunction[Any, a]): a
}
```

The state of a mailbox consists of a multi-set of messages. Messages are added to the mailbox the send method. Messages are removed using the receive method, which is passed a message processor f as argument, which is a partial function from messages to some arbitrary result type. Typically, this function is implemented as a pattern matching expression. The receive method blocks until there is a message in the mailbox for which its message processor is defined. The matching message is then removed from the mailbox and the blocked thread is restarted by applying the message processor to the message. Both sent messages and receivers are ordered in time. A receiver r is applied to a matching message m only if there is no other {message, receiver} pair which precedes m, r in the partial ordering on pairs that orders each component in time.

As a simple example of how mailboxes are used, consider a one-place buffer:

```
class OnePlaceBuffer {
  private val m = new MailBox;            // An internal mailbox
  private case class Empty, Full(x: int); // Types of messages we deal with
  m send Empty;                           // Initialization
  def write(x: int)
    { m receive { case Empty => m send Full(x) } }
  def read: int =
    m receive { case Full(x) => m send Empty ; x }
}
```

Here's how the mailbox class can be implemented:

```
class MailBox {
  private abstract class Receiver extends Signal {
    def isDefined(msg: Any): boolean
    var msg = null
}
```

We define an internal class for receivers with a test method `isDefined`, which indicates whether the receiver is defined for a given message. The receiver inherits from class `Signal` a `notify` method which is used to wake up a receiver thread. When the receiver thread is woken up, the message it needs to be applied to is stored in the `msg` variable of `Receiver`.

```
private val sent = new LinkedList[Any]
private var lastSent = sent
private val receivers = new LinkedList[Receiver]
private var lastReceiver = receivers
```

The mailbox class maintains two linked lists, one for sent but unconsumed messages, the other for waiting receivers.

```
def send(msg: Any) = synchronized {
  var r = receivers, r1 = r.next
  while (r1 != null && !r1.elem.isDefined(msg)) {
    r = r1; r1 = r1.next
  }
  if (r1 != null) {
    r.next = r1.next; r1.elem.msg = msg; r1.elem.notify
  } else {
    lastSent = insert(lastSent, msg)
  }
}
```

The `send` method first checks whether a waiting receiver is applicable to the sent message. If yes, the receiver is notified. Otherwise, the message is appended to the linked list of sent messages.

```
def receive[a](f: PartialFunction[Any, a]): a = {
  val msg: Any = synchronized {
    var s = sent, s1 = s.next
    while (s1 != null && !f.isDefinedAt(s1.elem)) {
      s = s1; s1 = s1.next
    }
    if (s1 != null) {
      s.next = s1.next; s1.elem
    } else {
      val r = insert(lastReceiver, new Receiver {
        def isDefined(msg: Any) = f.isDefinedAt(msg)
      })
      lastReceiver = r
      r.elem.wait()
      r.elem.msg
      }
    }
    f(msg)
  }
```

The `receive` method first checks whether the message processor function `f` can be applied to a message that has already been sent but that was not yet consumed. If yes, the thread continues immediately by applying `f` to the message. Otherwise, a new receiver is created and linked into the `receivers` list, and the thread waits for a notification on this receiver. Once the thread is woken up again, it continues by applying `f` to the message that was stored in the receiver. The insert method on linked lists is defined as follows.

```
  def insert(l: LinkedList[a], x: a): LinkedList[a] = {
    l.next = new LinkedList[a]
    l.next.elem = x
    l.next.next = l.next
    l
  }
```

The mailbox class also offers a method `receiveWithin` which blocks for only a specified maximal amount of time. If no message is received within the specified time interval (given in milliseconds), the message processor argument *f* will be unblocked with the special `TIMEOUT` message. The implementation of `receiveWithin` is quite similar to `receive`:

```
  def receiveWithin[a](msec: long)(f: PartialFunction[Any, a]): a = {
    val msg: Any = synchronized {
      var s = sent, s1 = s.next
      while (s1 != null && !f.isDefinedAt(s1.elem)) {
        s = s1; s1 = s1.next
      }
      if (s1 != null) {
        s.next = s1.next; s1.elem
      } else {
        val r = insert(lastReceiver, new Receiver {
            def isDefined(msg: Any) = f.isDefinedAt(msg)
        })
        lastReceiver = r
        r.elem.wait(msec)
        if (r.elem.msg == null) r.elem.msg = TIMEOUT
        r.elem.msg
      }
    }
    f(msg)
  }
```

```
} // end MailBox
```

The only differences are the timed call to `wait`, and the statement following it.

16.11 Actors

Chapter 2 sketched as a program example the implementation of an electronic auction service. This service was based on high-level actor processes that work by inspecting messages in their mailbox using pattern matching. A refined and optimized implementation of actors is found in the `scala.actors` package. We now give a sketch of a simplified version of the actors library.

The code below is different from the implementation in the `scala.actors` package, so it should be seen as an example how a simple version of actors could be implemented. It is not a description how actors are actually defined and implemented in the standard Scala library. For the latter, please refer to the Scala API documentation.

A simplified actor is just a thread whose communication primitives are those of a mailbox. Such an actor can be defined as a mixin composition extension of Java's standard `Thread` class with the `MailBox` class. We also override the `run` method of the `Thread` class, so that it executes the behavior of the actor that is defined by its `act` method. The `!` method simply calls the `send` method of the `MailBox` class:

```
abstract class Actor extends Thread with MailBox {
  def act(): unit
  override def run(): unit = act()
  def !(msg: Any) = send(msg)
}
```

III The Scala Language Specification

Version 1.0

Preface

Scala is a Java-like programming language which unifies object-oriented and functional programming. It is a pure object-oriented language in the sense that every value is an object. Types and behavior of objects are described by classes. Classes can be composed using mixin composition. Scala is designed to work seamlessly with two less pure but mainstream object-oriented languages – Java and C#.

Scala is a functional language in the sense that every function is a value. Nesting of function definitions and higher-order functions are naturally supported. Scala also supports a general notion of pattern matching which can model the algebraic types used in many functional languages.

Scala has been designed to interoperate seamlessly with Java (an alternative implementation of Scala also works for .NET). Scala classes can call Java methods, create Java objects, inherit from Java classes and implement Java interfaces. None of this requires interface definitions or glue code.

Scala has been developed from 2001 in the programming methods laboratory at EPFL. Version 1.0 was released in November 2003. This document describes the second version of the language, which was released in March 2006. It acts a reference for the language definition and some core library modules. It is not intended to teach Scala or its concepts; for this there are other documents [Oa04, Ode06, OZ05b, OCRZ03b, OZ05a].

Scala has been a collective effort of many people. The design and the implementation of version 1.0 was completed by Philippe Altherr, Vincent Cremet, Gilles Dubochet, Burak Emir, Stéphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, Matthias Zenger, and the author. Iulian Dragos, Gilles Dubochet, Sean McDirmid and Lex Spoon joined in the effort to develop the second version of the language and tools. Gilad Bracha, Craig Chambers, Erik Ernst, Matthias Felleisen, Shriram Krishnamurti, Gary Leavens, Sebastian Maneth, Erik Meijer, Klaus Ostermann, Didier Rémy, Mads Torgersen, and Philip Wadler have shaped the design of the language through lively and inspiring discussions and comments on previous versions of this document. The contributors to the Scala mailing list have also given very useful feedback that helped us improve the language and its tools.

Chapter 17

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

17.1 Identifiers

Syntax:

```
op        ::=  opchar {opchar}
varid     ::=  lower idrest
plainid   ::=  upper idrest
            |  varid
            |  op
id        ::= plainid
            |  ‘\‘’string chars‘\‘’
idrest    ::= {letter | digit} [‘_’ op]
```

There are three ways to form an identifier. First, an identifier can start with a letter which can be followed by an arbitrary sequence of letters and digits. This may be followed by underscore ‘_’ characters and another string composed of either letters and digits or of operator characters. Second, an identifier can start with an operator character followed by an arbitrary sequence of operator characters. The preceding two forms are called *plain* identifiers. Finally, an identifier may also be formed by an arbitrary string between back-quotes (host systems may impose some restrictions on which strings are legal for identifiers). The identifier then is composed of all characters excluding the backquotes themselves.

As usual, a longest match rule applies. For instance, the string

```
big_bob++=‘def‘
```

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

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

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

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

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

Example 17.1.1 Here are examples of identifiers:

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

Example 17.1.2 Backquote-enclosed strings are a solution when one needs to access Java identifiers that are reserved words in Scala. For instance, the statement `Thread.yield()` is illegal, since **`yield`** is a reserved word in Scala. However, here's a work-around:

```
Thread.`yield`()
```

17.2 Newline Characters

Syntax:

```
semi ::= ';' |  nl {nl}
```

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

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

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

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

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

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

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

Multiple statements are allowed in:

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

Multiple statements are disabled in:

1. the interval between matching `(` and `)` parenthesis tokens, except for nested regions where newlines are enabled, and
2. the interval between matching `[` and `]` bracket tokens, except for nested regions where newlines are enabled.
3. The interval between a **`case`** token and its matching => token, except for nested regions where newlines are enabled.
4. Any regions analyzed in XML mode (§17.5).

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

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

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

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

– between the condition of an conditional expression (§22.15) or while loop (§22.16) and the next following expression,
– between the enumerators of a for-comprehension (§22.18) and the next following expression, and
– after the initial **`type`** keyword in a type definition or declaration (§20.3).

A single new line token is accepted

– in front of an opening brace "{", if that brace is a legal continuation of the current statement or expression,
– after an infix operator, if the first token on the next line can start an expression (§22.11),

– in front of a parameter clause (§20.6), and
– after an annotation (§27).

Example 17.2.1 The following code contains four well-formed statements, each on two lines. The newline tokens between the two lines are not treated as statement separators.

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

while (x > 0)
  x  = x / 2

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

type
  IntList = List[int]
```

Example 17.2.2 The following code designates an anonymous class

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

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

```
new Iterator[int]

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

Example 17.2.3 The following code designates a single expression:

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

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

```
x < 0 ||

x > 10
```

Example 17.2.4 The following code designates a single, curried function definition:

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

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

```
def func(x: int)

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

Example 17.2.5 The following code designates an attributed definition:

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

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

```
@serializable

protected class Data { ... }
```

17.3 Literals

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

Syntax:

```
Literal       ::=  integerLiteral
                |  floatingPointLiteral
                |  booleanLiteral
                |  characterLiteral
                |  stringLiteral
                |  symbolLiteral
```

17.3.1 Integer Literals

Syntax:

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

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

However, if the expected type *pt* (§22) of a literal in an expression is either `byte`, `short`, or `char` and the integer number fits in the numeric range defined by the type, then the number is converted to type *pt* and the literal's type is *pt*. The numeric ranges given by these types are:

`byte`	-2^7 to 2^7-1
`short`	-2^{15} to $2^{15}-1$
`char`	0 to $2^{16}-1$

Example 17.3.1 Here are some integer literals:

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

17.3.2 Floating Point Literals

Syntax:

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

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

Example 17.3.2 Here are some floating point literals:

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

17.3.3 Boolean Literals

Syntax:

```
booleanLiteral      ::= true | false
```

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

17.3.4 Character Literals

Syntax:

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

A character literal is a single character enclosed in quotes. The character is either a printable unicode character or is described by an escape sequence (§17.3.6).

Example 17.3.3 Here are some character literals:

```
’a’    ’\u0041’    ’\n’    ’\t’
```

Note that ’`\u000A`’ is *not* a valid character literal because Unicode conversion is done before literal parsing and the Unicode character `\u000A` (line feed), and is not a printable character. One can use instead the escape sequence ‘`\n`’ or the octal escape ‘`\12`’ (§17.3.6).

17.3.5 String Literals

Syntax:

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

A string literal is a sequence of characters in double quotes. The characters are either printable unicode character or are described by escape sequences (§17.3.6). If the string literal contains a double quote character, it must be escaped, i.e. `\"`. The value of a string literal is an instance of class `String`.

Example 17.3.4 Here are some string literals:

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

Multi-Line String Literals

Syntax:

```
stringLiteral  ::=  '"""' multiLineChars '"""'
multiLineChars ::=  {['"'] ['"'] charNoDoubleQuote}
```

A multi-line string literal is a sequence of characters enclosed in triple quotes `""" ... """`. The sequence of characters is arbitrary, except that it may not contain a triple quote. Characters must not necessarily be printable; newlines or other control characters are also permitted. Unicode escapes work as everywhere else, but none of the escape sequences in (§17.3.6) is interpreted.

Example 17.3.5 Here is a multi-line string literal:

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

This would produce the string:

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

The Scala library contains a utility method `stripMargin` which can be used to strip leading whitespace from multi-line strings. The expression

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

evaluates to

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

Method `stripMargin` is defined in class `scala.runtime.RichString`. Because there is a predefined implicit conversion (§22.24) from `String` to `RichString`, the method is applicable to all strings.

17.3.6 Escape Sequences

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

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

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

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

17.3.7 Symbol literals

Syntax:

```
symbolLiteral     ::= ‘’’ idrest
```

A symbol literal `'`x is a shorthand for the expression `scala.Symbol("`x`").intern`. `Symbol` is a case class (§21.3.2), which is defined as follows.

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

The `intern` method turns symbols into unique references: If two interned symbols have the same name, then they must be the same object.

17.4 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.

17.5 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 successfully parsed. In this case, since code and XML fragments can be nested, the parser has to maintain a stack that reflects the nesting of XML and Scala expressions adequately.

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

Example 17.5.1 The following value definition uses an XML literal with two embedded Scala expressions

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

Chapter 18

Identifiers, Names and Scopes

Names in Scala identify types, values, methods, and classes which are collectively called *entities*. Names are introduced by local definitions and declarations (§20), inheritance (§21.1.3), import clauses (§20.7), or package clauses (§25.2) which are collectively called *bindings*.

Bindings of different kinds have a precedence defined on them: Definitions (local or inherited) have highest precedence, followed by explicit imports, followed by wildcard imports, followed by package members, which have lowest precedence.

There are two different name spaces, one for types (§19) and one for terms (§22). The same name may designate a type and a term, depending on the context where the name is used.

A binding has a *scope* in which the entity defined by a single name can be accessed using a simple name. Scopes are nested. A binding in some inner scope *shadows* bindings of lower precedence in the same scope as well as bindings of the same or lower precedence in outer scopes.

Note that shadowing is only a partial order. In a situation like

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

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

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

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

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

Example 18.0.2 Assume the following two definitions of a objects named `X` in packages `P` and `Q`.

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

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

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

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

A reference to a qualified (type- or term-) identifier $e.x$ refers to the member of the type T of e which has the name x in the same namespace as the identifier. It is an error if T is not a value type (§19.2). The type of $e.x$ is the member type of the referenced entity in T.

Chapter 19

Types

Syntax:

```
Type              ::=  InfixType [‘=>’ Type]
                    |  ‘(’ [‘=>’ Type] ‘)’ ‘=>’ Type
InfixType         ::=  CompoundType {id [nl] CompoundType}
CompoundType      ::=  AnnotType {with AnnotType} [Refinement]
AnnotType         ::=  {Annotation} SimpleType
SimpleType        ::=  SimpleType TypeArgs
                    |  SimpleType ‘#’ id
                    |  StableId
                    |  Path ‘.’ type
                    |  ‘(’ Types [‘,’] ’)’
TypeArgs          ::=  ‘[’ Types ‘]’
Types             ::=  Type {‘,’ Type}
```

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

Every concrete value type can be represented as a *class type*, i.e. a type designator (§19.2.3) that refers to a class[1] (§21.3), or as a *compound type* (§19.2.7) representing an intersection of types, possibly with a refinement (§19.2.7) that further constrains the types of its members. Abstract value types are introduced by type parameters (§20.4) and abstract type bindings (§20.3). Parentheses in types are used for grouping.

Non-value types capture properties of identifiers that are not values (§19.3). For example, a type constructor (§19.3.3) does not directly specify the type of values. However, when a type constructor is applied to the correct type arguments, it yields

[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).

a first-order type, which may be a value type.

Non-value types are expressed indirectly in Scala. E.g., a method type is described by writing down a method signature, which in itself is not a real type, although it gives rise to a corresponding function type (§19.3.1). Type constructors are another example, as one can write **type** `Swap[m[_, _], a,b] = m[b, a]`, but there is **no** syntax to write the corresponding anonymous type function directly.

19.1 Paths

Syntax:

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

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

A path is one of the following.

- The empty path ϵ (which cannot be written explicitly in user programs).
- C`.`**`this`**, where C references a class. The path **`this`** is taken as a shorthand for C`.`**`this`** where C is the name of the class directly enclosing the reference.
- p`.`x where p is a path and x is a stable member of p. *Stable members* are 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 parent class M of C. The prefix **`super`** is taken as a shorthand for C`.`**`super`** where C is the name of the class directly enclosing the reference.

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

19.2 Value Types

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

19.2.1 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 (§22) to `scala.AnyRef`. The type denotes the set of values consisting of **null** and the value denoted by p.

19.2.2 Type Projection

Syntax:

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

A type projection $T\#x$ references the type member named x of type T. If x references an abstract type member, then T must be a singleton type.

19.2.3 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 (§19.1) and t is a type name. Such a type designator is equivalent to the type projection p**.type**$\#x$.

Example 19.2.1 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
```

19.2.4 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 $a_1, \ldots, a_n$.

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

Example 19.2.2 Given the partial type definitions:

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

the following parameterized types are well formed:

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

Example 19.2.3 Given the type definitions of Example 19.2.2, the following types are ill-formed:

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

19.2.5 Tuple Types

Syntax:

```
SimpleType    ::=    '(' Types [','] ')'
```

A tuple type $(T_1, \ldots, T_n)$ is an alias for the class `scala.Tuple`$n[T_1, \ldots, T_n]$, where $n \geq 2$. The type may also be written with a trailing comma, i.e. $(T_1, \ldots, T_n,)$. The unary tuple type `scala.Tuple1[T]` can be written in tuple syntax only by using a trailing comma, i.e. $(T,)$.

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

```
case class Tuplen[+T1, ..., +Tn](_1: T1, ..., _n: Tn)
extends Productn[T1, ..., Tn] {}

trait Productn[+T1, +T2, +Tn] {
  override def arity = n
  def _1: T1
  ...
  def _n:Tn
}
```

19.2.6 Annotated Types

Syntax:

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

An annotated type `@`a_1 … `@`a_n T attaches annotations $a_1, \ldots, a_n$ to the type T (§27).

19.2.7 Compound Types

Syntax:

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

A compound type T_1 **`with`** … **`with`** T_n $\{R\}$ represents objects with members as given in the component types $T_1, \ldots, T_n$ and the refinement $\{R\}$. A refinement $\{R\}$ contains declarations and type definitions. 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 (§21.1.4) 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 `{}`.

19.2.8 Infix Types

Syntax:

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

An infix type T_1 *op* T_2 consists of an infix operator *op* which gets applied to two type operands T_1 and T_2. The type is equivalent to the type application $op[T_1, T_2]$. The infix operator *op* may be an arbitrary identifier, except for `*`, which is reserved as a postfix modifier denoting a repeated parameter type (§20.6.2).

All type infix operators have the same precedence; parentheses have to be used for grouping. The associativity (§22.11) of a type operator is determined as for term operators: type operators ending in a colon ‘`:`’ are right-associative; all other operators are left-associative.

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

19.2.9 Function Types

Syntax:

```
Type   ::=  InfixType '=>' Type
         |  '(' ['=>' Type] ')' '=>' 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. The type `(=>`T`) =>` U represents functions with call-by-name parameters (§20.6.1) of type T which yield results of type 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 (§20.5) in their result type and contravariant in their argument types.

A call-by-name function type `(=>` T`) =>` U is a shorthand for the class type `ByNameFunction[`T`,`U`]`, which is defined as follows.

```
package scala
trait ByNameFunction[-T, +R] {
  def apply(x: => T): R
  override def toString() = "<function>"
}
```

19.2.10 Primitive Types Defined in *Predef*

The object `Predef` is imported implicitly into every Scala program . It contains type definitions which establish the primitive types mentioned above as aliases of class types. Numeric and boolean types are equated with standard Scala classes. The

String type is equated with the string class of the underlying host system. In a Java environment, Predef contains the following bindings, among others:

```
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 String  = java.lang.String
```

19.3 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.

19.3.1 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.

We let 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. 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 (§22.24).

Example 19.3.1 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
```

19.3.2 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 (§19.2.4) 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 19.3.2 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 >: Nothing <: Any] List[a]
union : [a >: Nothing <: Comparable[a]] (x: Set[a], xs: Set[a]) Set[a]  .
```

19.3.3 Type Constructors

A type constructor is represented internally much like a polymorphic method type. `[`$\pm$ a_1 `>:` L_1 `<:` $U_1, \ldots, \pm a_n$ `>:` L_n `<:` U_n`]` T represents a type that is expected by a type constructor parameter (§20.4) or an abstract type constructor binding (§20.3) with the corresponding type parameter clause.

Example 19.3.3 Consider this fragment of the `Iterable[+x]` class:

```
trait Iterable[+x] {
  def flatMap[newType[+x] <: Iterable[x], s](f: t => newType[s]): newType[s]
}
```

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

19.4 Base Types and Member Definitions

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

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

These notions are defined mutually recursively as follows.

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

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

2. The notion of a type T *in class* C *seen from some prefix type* S makes sense only if the prefix type S has a type instance of class C as a base type, say $S'\#C[T_1, \ldots, T_n]$. Then we define as follows.

- If $S = \epsilon$**.type**, then T in C seen from S is T itself.
- Otherwise, if T is the i'th type parameter of some class D, then
  - If S has a base type $D[U_1, \ldots, U_n]$, for some type parameters $[U_1, \ldots, U_n]$, then T in C seen from S is U_i.
  - Otherwise, if C is defined in a class C', then T in C seen from S is the same as T in C' seen from S'.
  - Otherwise, if C is not defined in another class, then T in C seen from S is T itself.
- Otherwise, if T is the singleton type D**.this.type** for some class D then
  - If D is a subclass of C and S has a type instance of class D among its base types, then T in C seen from S is S.

  - Otherwise, if C is defined in a class C', then T in C seen from S is the same as T in C' seen from S'.
  - Otherwise, if C is not defined in another class, then T in C seen from S is T itself.

- If T is some other type, then the described mapping is performed to all its type components.

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

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

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

19.5 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.

19.5.1 Type Equivalence

Equivalence ($\equiv$) 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** $\equiv$ q.**type**.
- If O is defined by an object definition, and p is a path consisting only of package or object selectors and ending in O, then O.**this.type** $\equiv$ p.**type**.
- Two compound types (§19.2.7) are equivalent if the sequences of their component are pairwise equivalent, and occur in the same order, and their refinements are equivalent. Two refinements are equivalent if they bind the same names and the modifiers, types and bounds of every declared entity are equivalent in both refinements.

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

- Two method types (§19.3.1) are equivalent if they have equivalent result types, both have the same number of parameters, and corresponding parameters have equivalent types. Note that the names of parameters do not matter for method type equivalence.
- Two polymorphic method types (§19.3.2) 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 type constructors (§19.3.3) 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 variances, lower and upper bounds of corresponding type parameters are equivalent.

19.5.2 Conformance

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

- Conformance includes equivalence. If $T \equiv U$ then $T <: U$.
- For every value type T, `scala.Nothing` <: T <: `scala.Any`.
- For every type constructor T (with any number of type parameters), `scala.Nothing` <: T <: `scala.Any`.
- For every class type T <: `scala.AnyRef` one has `scala.Null` <: T.
- A type variable or abstract type t conforms to its upper bound and its lower bound conforms to t.
- A class type or parameterized type conforms to any of its base-types.
- A singleton type p`.type` conforms to the type of the path p.
- A 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** … **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 d of a type or value x in R there exists a member binding of x in T which subsumes d, then T conforms to the compound type U_1 **with** … **with** U_n $\{R\}$.

- 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 $(T'_1, \ldots, T'_n)U'$.
- The polymorphic type $[a_1 >: L_1 <: U_1, \ldots, a_n >: L_n <: U_n]T$ conforms to the polymorphic type $[a_1 >: L'_1 <: U'_1, \ldots, a_n >: L'_n <: U'_n]T'$ if, assuming $L'_1 <: a_1 <: U'_1, \ldots, L'_n <: a_n <: U'_n$ one has $T <: T'$ and $L_i <: L'_i$ and $U'_i <: U_i$ for $i = 1, \ldots, n$.
- Type constructors T and T' follow a similar discipline. We characterize T and T' by their type parameter clauses $[a_1, \ldots, a_n]$ and $[a'_1, \ldots, a'_n]$, where an a_i or a'_i may include a variance annotation, a higher-order type parameter clause, and bounds. Then, T conforms to T' if any list $[t_1, \ldots, t_n]$ – with declared variances, bounds and higher-order type parameter clauses – of valid type arguments for T' is also a valid list of type arguments for T and $T[t_1, \ldots, t_n] <: T'[t_1, \ldots, t_n]$. Note that this entails that:
  - The bounds on a_i must be weaker than the corresponding bounds declared for a'_i.
  - The variance of a_i must match the variance of a'_i, where covariance matches covariance, contravariance matches contravariance and any variance matches invariance.
  - Recursively, these restrictions apply to the corresponding higher-order type parameter clauses of a_i and a'_i.

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

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

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

Note. The least upper bound or greatest lower bound of a set of types does not always exist. For instance, consider the class definitions

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

Then the types `A[Any], A[A[Any]], A[A[A[Any]]], ...` form a descending sequence of upper bounds for `B` and `C`. The least upper bound would be the infinite limit of that sequence, which does not exist as a Scala type. Since cases like this are in general impossible to detect, a Scala compiler is free to reject a term which has a type specified as a least upper or greatest lower bound, and that bound would be more complex than some compiler-set limit[3].

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

19.6 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 an alias type is the erasure of its right-hand side.
- The erasure of an abstract type is the erasure of its upper bound.
- The erasure of the parameterized type `scala.Array`$[T_1]$ is `scala.Array`$[|T_1|]$.
- The erasure of every other parameterized type $T[T_1, \ldots, T_n]$ is $|T|$.
- The erasure of a singleton type p`.type` is the erasure of the type of p.
- The erasure of a type projection $T\#x$ is $|T|\#x$.
- The erasure of a compound type T_1 `with` ... `with` T_n $\{R\}$ is $|T_1|$.
- The erasure of every other type is the type itself.

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

Chapter 20

Basic Declarations and Definitions

Syntax:

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

A *declaration* introduces names and assigns them types. It can form part of a class definition (§21.1) or of a refinement in a compound type (§19.2.7).

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

The scope of a name introduced by a declaration or definition is the whole statement sequence containing the binding. However, there is a restriction on forward references in blocks: In a statement sequence $s_1 \ldots s_n$ making up a block, if a simple name in s_i refers to an entity defined by s_j where $j \geq i$, then none of the definitions between and including s_i and s_j may be a value or variable definition.

20.1 Value Declarations and Definitions

Syntax:

```
Dcl          ::=  val ValDcl
```

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

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

A value definition **val** x: T = e defines x as a name of the value that results from the evaluation of e. 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 (§22).

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 (§24.1) 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 => {x_1, ..., x_n}}
val x_1 = $x._1
...
val x_n = $x._n  .
```

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

2. If p has a unique bound variable x:

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

3. If p has no bound variables:

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

Example 20.1.1 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 => {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** $p_n : T = e$.

20.2 Variable Declarations and Definitions

Syntax:

```
Dcl           ::=  var VarDcl
Def           ::=  var VarDef
VarDcl        ::=  ids ‘:’ Type
VarDef        ::=  ids [‘:’ Type] ‘=’ Expr
                |  ids ‘:’ Type ‘=’ ‘_’
```

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

```
def x: T
def x_= (y: T): unit
```

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

A variable definition **var** x: $T = e$ introduces a mutable variable with type T and initial value as given by the expression e. The type T can be omitted, in which case the type of e is assumed. If T is given, then e is expected to conform to it (§22).

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

`0`	if T is `int` or one of its subrange types,
`0L`	if T is `long`,
`0.0f`	if T is `float`,
`0.0d`	if T is `double`,
`false`	if T is `boolean`,
`{}`	if T is `unit`,
`null`	for all other types T.

When they occur as members of a template, both forms of variable definition also introduce a getter function x which returns the value currently assigned to the variable, as well as a setter function x_= which changes the value currently assigned to the variable. The functions have the same signatures as for a variable declaration.

The template then has these getter and setter functions as members, whereas the original variable cannot be accessed directly as a template member.

Example 20.2.1 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
  private var m: int = 0
  private var s: int = 0

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

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

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

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

20.3 Type Declarations and Type Aliases

Syntax:

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

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

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

A type constructor declaration imposes additional restrictions on the concrete types for which t may stand. Besides the bounds L and U, the type parameter clause may impose higher-order bounds and variances, as governed by the conformance of type constructors (§19.5.2).

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

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

A *type alias* **type** t = T defines t to be an alias name for the type T. The left hand side of a type alias may have a type parameter clause, e.g. **type** t[*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 (§20) and type parameters (§20.6) 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 20.3.1 The following are legal type declarations and definitions:

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

The following are illegal:

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

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

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

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 20.3.2 The Predef object 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. Furthermore, because Tuple2 is a case class (§21.3.2), Pair2 is also an alias for the case class factory Tuple2, and this holds for in expressions as well as patterns. Hence, the following are all legal uses of Pair.

```
val x: Pair[int, String] = new Pair(1, "abc")
val y: Pair[String, int] = x match {
  case Pair(i, s) => Pair(z + i, i * i)
}
```

20.4 Type Parameters

Syntax:

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

Type parameters appear in type definitions, class definitions, and function definitions. In this section we consider only type parameter definitions with lower bounds >: L and upper bounds <: U whereas a discussion of view bounds <% U is deferred to Section 23.4.

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

The names of all type parameters must be pairwise different in their enclosing type parameter clause. The scope of a type parameter includes in each case the whole type parameter clause. Therefore it is possible that a type parameter appears as part of its own bounds or the bounds of other type parameters in the same clause. However, a type parameter may not be bounded directly or indirectly by itself.

A type constructor parameter adds a nested type parameter clause to the type parameter. The most general form of a type constructor parameter is $\pm$ *t*[*tps*] >: *L* <: *U*.

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

Example 20.4.1 Here are some well-formed type parameter clauses:

```
[s, t]
[ex <: Throwable]
[a <: Comparable[b], b <: a]
[a, b >: a, c >: a <: b]
[m[x], n[x]]
[m[_], n[_]] // equivalent to previous clause
[m[x <: bound[x]], bound[_]]
[m[+x] <: Iterable[x]]
```

The following type parameter clauses are illegal:

```
[a >: a]                 // illegal, 'a' has itself as bound
[a <: b, b <: c, c <: a] // illegal, 'a' has itself as bound
[a, b, c >: a <: b]      // illegal lower bound 'a' of 'c' does
                         // not conform to upper bound 'b'.
```

20.5 Variance Annotations

Variance annotations indicate how instances of parameterized types vary with respect to subtyping (§19.5.2). 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, or a type declaration **type** t[*tps*] >: L <: U type parameters labeled '+' must only appear in covariant position whereas type parameters labeled '–' must only appear in contravariant position. Analogously, for a class definition **class** c[*tps*](*ps*) **requires** 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 $S[\dots T \dots]$: 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 $S[\dots T \dots]$.

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

Example 20.5.1 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](x: a, y: b) {
  var fst: a = x          // **** error: illegal variance:
  var snd: b = y          // ‘a’, ‘b’ occur in invariant position.
}
```

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

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

Example 20.5.2 The following variance annotation is illegal, since a appears in contravariant position in the parameter of append:

```
abstract class 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:

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

Example 20.5.3 Here is a case where a contravariant type parameter is useful.

```
abstract class OutputChannel[-a] {
  def write(x: a): unit
}
```

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

20.6 Function Declarations and Definitions

Syntax:

```
Dcl                  ::=  def FunDcl
FunDcl               ::=  FunSig ‘:’ Type
Def                  ::=  def FunDef
```

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

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.

A type parameter clause *tps* consists of one or more type declarations (§20.3), 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.

20.6.1 By-Name Parameters

Syntax:

```
ParamType         ::=  ‘=>’ Type
```

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

Example 20.6.1 The declaration

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

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

20.6.2 Repeated Parameters

Syntax:

```
ParamType          ::=  Type '*'
```

The last value parameter of a parameter section may be suffixed by "*", e.g. `(..., ` x`:`T`*)`. The type of such a *repeated* parameter inside the method is then the sequence type `scala.Seq[`T`]`. Methods with repeated parameters $T*$ take a variable number of arguments of type T. That is, if a method m with type $(T_1, \ldots, T_n, S*)U$ 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. The only exception to this rule is if the last argument is marked to be a *sequence argument* via a `_*` type annotation. If m above is applied to arguments $(e_1, \ldots, e_n, e'$`: _*)`, then the type of m in that application is taken to be $(T_1, \ldots, T_n,$ `scala.Seq[`S`])`.

Example 20.6.2 The following method definition computes the sum of a variable number of integer arguments.

```
def sum(args: int*) = {
  var result = 0
  for (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)
```

Furthermore, assume the definition:

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

The following applications method `sum` is ill-formed:

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

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

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

20.6.3 Procedures

Syntax:

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

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

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

Example 20.6.3 Here is a declaration and a definition of a procedure named `write`:

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

The code above is implicitly completed to the following code:

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

20.6.4 Method Return Type Inference

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

Example 20.6.4 Assume the following definitions:

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

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

20.7 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 (§19.1) 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.

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

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

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

Example 20.7.1 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 21

Classes and Objects

Syntax:

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

Classes (§21.3) and objects (§21.4) are both defined in terms of *templates*.

21.1 Templates

Syntax:

```
ClassTemplate   ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate   ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents    ::=  Constr {with AnnotType}
TraitParents    ::=  AnnotType {with AnnotType}
TemplateBody    ::=  [nl] ‘{’ [id [‘:’ Type] ‘=>’]
                     TemplateStat {semi TemplateStat} ‘}’
```

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

Each trait reference mt_i must denote a trait (§21.3.3). By contrast, the superclass constructor sc normally refers to a class which is not a trait. It is possible to write a list of parents that starts with a trait reference, e.g. mt_1 **with** … **with** mt_n. In that case the list of parents is implicitly extended to include the supertype of mt_1 as first parent type. The new supertype must have at least one constructor that does not take parameters. In the following, we will always assume that this implicit extension has been performed, so that the first parent class of a template is a regular superclass constructor, not a trait reference.

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

sc **with** mt_1 **with** ... **with** mt_n $\{stats\}$

becomes

mt_1 **with** ... **with** mt_n $\{stats\}$ **with** `ScalaObject` $\{stats\}$.

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

The *least proper supertype* of a template is the class type or compound type (§19.2.7) consisting of all its parent class types.

The statement sequence *stats* contains member definitions that define new members or overwrite members in the parent classes. If the template forms part of a class definition, the statement part *stats* may also contain declarations of abstract members. Furthermore, *stats* may contain expressions that are executed in the order they are given as part of the initialization of a template.

The sequence of template statements may be prefixed with a formal parameter definition and an arrow, e.g. x `=>`, or x`:` T `=>`. If a formal parameter is given, it can be used as an alias for the reference **this** throughout the body of the template. If the formal parameter comes with a type T, this type is assumed to be the *self-type* (§21.3) of the underlying class.

Example 21.1.1 Consider the following class definitions:

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

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

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

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

Template Evaluation. Consider a template sc **with** mt_1 **with** mt_n {*stats*}.

If this is the template of a trait (§21.3.3) then its *mixin-evaluation* consists of an evaluation of the statement sequence *stats*.

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

- First, the superclass constructor *sc* is evaluated (§21.1.1).
- Then, all base classes in the template's linearization (§21.1.2) up to the template's superclass denoted by *sc* are mixin-evaluated. Mixin-evaluation happens in reverse order of occurrence in the linearization, i.e. the class immediately preceding *sc* is evaluated first.
- Finally the statement sequence *stats* is evaluated.

21.1.1 Constructor Invocations

Syntax:

```
    Constr  ::=  AnnotType {'(' [Exprs [',']] ')'}
```

Constructor invocations define the type, members, and initial state of objects created by an instance creation expression, or of parts of an object's definition which are inherited by a class or object definition. A constructor invocation is a function application $x.c[targs](args_1)...(args_n)$, where x is a stable identifier (§19.1), c is a type name which either designates a class or defines an alias type for one, *targs* is a type argument list, and $args_1, \ldots, args_n$ are argument lists, which match the parameters of one the constructors of that class.

The prefix '`x.`' can be omitted. A type argument list can be given only if the class c takes type parameters. Even then it can be omitted, in which case a type argument list is synthesized using local type inference (§22.24.4). If no explicit arguments are given, an empty list `()` is implicitly supplied.

An evaluation of a constructor invocation $x.c[targs](args_1)...(args_n)$ consists of the following steps:

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

21.1.2 Class Linearization

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

Definition 21.1.2 Let C be a class with template C_1 **with** ... **with** C_n { *stats* }. The *linearization* of C, $\mathscr{L}(C)$ is defined as follows:

$$\mathscr{L}(C) \quad = \quad C\ ,\ \mathscr{L}(C_n)\ \vec{+}\ \ldots\ \vec{+}\ \mathscr{L}(C_1)$$

Here $\vec{+}$ denotes concatenation where elements of the right operand replace identical elements of the left operand:

$$\begin{array}{rcll} \{a, A\}\ \vec{+}\ B & = & a, (A\ \vec{+}\ B) & \textbf{if} a \notin B \\ & = & A\ \vec{+}\ B & \textbf{if} a \in B \end{array}$$

Example 21.1.3 Consider the following class definitions.

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

Then the linearization of class `Iter` is

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

Note that the linearization of a class refines the inheritance relation: if C is a subclass of D, then C precedes D in any linearization where both C and D occur. Definition 21.1.2 also satisfies the property that a linearization of a class always contains the linearization of its direct superclass as a suffix. For instance, the linearization of `StringIterator` is

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

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

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

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

21.1.3 Class Members

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

Definition 21.1.4 A member definition M *matches* a member definition M', if M and M' bind the same name, and one of following holds.

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

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

Definition 21.1.5 A *concrete member* of a class C is any concrete definition M in some class $C_i \in \mathcal{L}(C)$, except if there is a preceding class $C_j \in \mathcal{L}(C)$ where $j < i$ which directly defines a concrete member M' matching M.

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

This definition also determines the overriding relationships between matching members of a class C and its parents (§21.1.4). First, a concrete definition always overrides an abstract definition. Second, for definitions M and M' which are both concrete or both abstract, M overrides M' if M appears in a class that precedes (in the linearization of C) the class in which M' is defined.

It is an error if a template directly defines two matching members. It is also an error if a template contains two members (directly defined or inherited) with the same name and the same erased type (§19.6).

Example 21.1.6 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 defined abstract member `h`. It inherits member `f` from class `B` and member `g` from class `A`.

21.1.4 Overriding

A member M of class C that matches (§21.1.3) a non-private member M' of a base class of C is said to *override* that member. In this case the binding of the overriding member M must subsume (§19.5.2) the binding of the overridden member M'. Furthermore, the following restrictions on modifiers apply to M and M':

- M' must not be labeled **`final`**.
- M must not be **`private`** (§21.2).
- If M is labeled **`private`**`[`C`]` for some enclosing class or package C, then M' must be labeled **`private`**`[`C'`]` for some class or package C' where C' equals C or C' is contained in C.
- If M is labeled **`protected`**, then M' must also be labeled **`protected`**.
- If M' is not an abstract member, then M must be labeled **`override`**.
- If M' is incomplete (§21.2) in C then M must be labeled **`abstract override`**.

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

Example 21.1.7 Consider the definitions:

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

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

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

21.1.5 Inheritance Closure

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

- If T is in $\mathscr{S}$, then every type T' which forms syntactically a part of T is also in $\mathscr{S}$.
- If T is a class type in $\mathscr{S}$, then all parents (§21.1) of T are also in $\mathscr{S}$.

It is a static error if the inheritance closure of a class type consists of an infinite number of types. (This restriction is necessary to make subtyping decidable [KP07]).

21.1.6 Early Definitions

Syntax:

```
EarlyDefs        ::= ‘{’ [EarlyDef {semi EarlyDef}] ‘}’ with
EarlyDef         ::=  Annotations Modifiers PatDef
```

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

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

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

An early definition is type-checked and evaluated in the scope which is in effect just before the template being defined, augmented by any type parameters of the enclosing class and by any early definitions preceding the one being defined. In particular, any reference to **this** in the right-hand side of an early definition refers to the identity of **this** just outside the template. Consequently, it is impossible that an early definition refers to the object being constructed by the template, or refers to one of its fields and methods, except for any other preceding early definition in the same section.

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

Example 21.1.8 Early definitions are particularly useful for traits, which do not have normal constructor parameters. Example:

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

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

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

21.2 Modifiers

Syntax:

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

Member definitions may be preceded by modifiers which affect the accessibility and usage of the identifiers bound by them. If several modifiers are given, their order does not matter, but the same modifier may not occur 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 or declaration in a template. Such members can be accessed only from within the directly enclosing template and its companion module or companion class (§Example 21.4.1). They are not inherited by subclasses and they may not override definitions in parent classes.

  The modifier can be *qualified* with an identifier C (e.g. **`private`**`[`C`]`) that must denote a class or package enclosing the definition. Members labeled with such a modifier are accessible respectively only from code inside the package C or only from code inside the class C and its companion module (§21.4). Such members are also inherited only from templates inside C.

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

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

Class-private or object-private members may not be deferred, and may not have **protected**, **final** or **override** modfiers.

- The **protected** modifier applies to class member definitions. Protected members of a class can be accessed from within

  - the template of the defining class,
  - all templates that have the defining class as a base class,
  - the companion module of any of those classes.

  A **protected** modifier can be qualified with an package identifier *C* (e.g. **protected**[*C*]) that must denote a class or package enclosing the definition. Members labeled with such a modifier are also accessible respectively from all code inside the package *C* or from all code inside the class *C* and its companion module (§21.4).

  A protected identifier *x* may be used as a member name in a selection *r*.*x* only if one of the following applies:

  - The access is within the template defining the member, or, if a qualification *C* is given, inside the package *C*, or the class *C*, or its companion module, or
  - *r* is one of the reserved words **this** and **super**, or
  - *r*'s type conforms to a type-instance of the class which contains the access.

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

- The **override** modifier applies to class member definitions or declarations. It is mandatory for member definitions or declarations that override some other concrete member definition in a parent class. If an **override** modifier is given, there must be at least one overridden member definition or declaration (either concrete or abstract).

- The **override** modifier has a different significance when combined with the **abstract** modifier. That modifier combination is only allowed for value members of traits. A member labeled **abstract override** must override at least one other member and all members overridden by it must be incomplete.

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

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

- The **abstract** modifier is used in class definitions. It is redundant for traits, and mandatory for all other classes which have incomplete members. Abstract classes cannot be instantiated (§22.9) with a constructor invocation unless followed by mixins and/or a refinement which override all incomplete members of the class. A case class (§21.3.2) cannot be **abstract**.

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

- The **final** modifier applies to class member definitions and to class definitions. A **final** class member definition may not be overridden in subclasses. A **final** class may not be inherited by a template. **final** is redundant for object definitions. Members of final classes or objects are implicitly also final, so the **final** modifier is 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 directly inherited, except if the inheriting template is defined the same source file as the inherited class. However, subclasses of a sealed class can inherited anywhere.

Example 21.2.1 The following code illustrates the use of qualified private:

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

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

Example 21.2.2 A useful idiom to prevent clients of a class from constructing new instances of that class is to declare the class **abstract** and **sealed**:

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

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

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

A similar access restriction can be achieved by marking the primary constructor **`private`** (see Example 21.3.2).

21.3 Class Definitions

Syntax:

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

The most general form of class definition is

$$\textbf{class}\ c[tps]\ as\ m(ps_1)\ldots(ps_n)\ \textbf{requires}\ s\ \textbf{extends}\ t \qquad (n \geq 0).$$

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*.

as is a possibly empty sequence of annotations (§27). If any annotations are given, they apply to the primary constructor of the class.

m is an access modifier (§21.2) such as **`private`** or **`protected`**, possibly with a qualification. If such an access modifier is given it applies to the primary constructor to the class.

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

If a formal parameter declaration $x : T$ is preceded by a **val** or **var** keyword, an accessor (getter) definition (§20.2) for this parameter is implicitly added to the class. The getter introduces a value member x of class c that is defined as an alias of the parameter. If the introducing keyword is **var**, a setter accessor *x*_= (§20.2) is also implicitly added to the class. In invocation of that setter *x*_=(*e*) changes the value of the parameter to the result of evaluating e. The formal parameter declaration may contain modifiers, which then carry over to the accessor definition(s). A formal parameter prefixed by **val** or **var** may not at the same time be a call-by-name parameter (§20.6.1).

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 **requires** 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 (§21.1) of the form

sc **with** mt_1 **with** ... **with** mt_m { *stats* } $(m \geq 0)$

which defines the base classes, behavior and initial state of objects of the class. The extends clause **extends** *sc* **with** mt_1 **with** ... **with** mt_m 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.

Example 21.3.1 The following example illustrates **val** and **var** parameters of a class `C`:

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

Example 21.3.2 The following class can be created only from its companion module.

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

21.3.1 Constructor Definitions

Syntax:

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

A class may have additional constructors besides the primary constructor. These are defined by constructor definitions of the form **def this**(ps_1)...(ps_n) = e. Such a definition introduces an additional constructor for the enclosing class, with parameters as given in the formal parameter lists $ps_1, \ldots, ps_n$, and whose evaluation is defined by the constructor expression e. The scope of each formal parameter is the constructor expression e. A constructor expression is either a self constructor invocation **this**($args_1$)...($args_n$) or a block which begins with a self constructor invocation. The self constructor invocation must construct a generic instance of the class. I.e. if the class in question has name C and type parameters [tps], then a self constructor invocation must generate an instance of C[tps]; it is not permitted to instantiate formal type parameters.

The signature and the self constructor invocation of a constructor definition are type-checked and evaluated in the scope which is in effect at the point of the enclosing class definition, augmented by any type parameters of the enclosing class and by any early definitions (§21.1.6) of the enclosing template. The rest of the constructor expression is type-checked and evaluated as a function body in the current class.

If there are auxiliary constructors of a class C, they form together with C's primary constructor (§21.3) an overloaded constructor definition. The usual rules for overloading resolution (§22.24.3) apply for constructor invocations of C, including for the self constructor invocations in the constructor expressions themselves. However, unlike other methods, constructors are never inherited. To prevent infinite cycles of constructor invocations, there is the restriction that every self constructor invocation must refer to a constructor definition which precedes it (i.e. it must refer to either a preceding auxiliary constructor or the primary constructor of the class).

Example 21.3.3 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 three constructors. The second constructor constructs an singleton list, while the third one constructs a list with a given head and tail.

21.3.2 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 formal parameters in the first parameter section of a case class are called *elements*; they treated specially. First, the value of such a parameter can be extracted as a field of a constructor pattern. Second, a **val** prefix is implicitly added to such a parameter, unless the parameter carries already a **val** or **var** modifier. Hence, an accessor definition for the parameter is generated (§21.3).

A case class definition of $c[tps](ps_1)\ldots(ps_n)$ 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_1)...(ps_n): s = **new** c[tps](xs_1)...(xs_n)

(Here, s is the self type of class c and each xs_i denotes the parameters of ps_i. If a type parameter section is missing in the class, it is also missing in the factory definition).

Every case class implicitly overrides some method definitions of class `scala.AnyRef` (§28.1) 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 both belong to the case class in question and they have equal (with respect to `equals`) constructor arguments.

Method `hashCode: ()int` computes a hash-code 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 elements.

Example 21.3.4 Here is the definition of abstract syntax for lambda calculus:

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

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

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

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

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

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

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

21.3.3 Traits

Syntax:

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

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

Assume a trait D defines some aspect of an instance x of type C (i.e. D is a base class of C). Then the *actual supertype* of D in x is the compound type consisting of all the base classes in $\mathscr{L}(C)$ that succeed D. The actual supertype gives the context for resolving a **super** reference in a trait (§22.4). Note that the actual supertype depends on the type to which the trait is added in a mixin composition; it is not statically known at the time the trait is defined.

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

Example 21.3.5 The following trait defines the property of being comparable to objects of some type. It contains an abstract method < and default implementations of the other comparison operators <=, >, and >=.

```
trait Comparable[t <: Comparable[t]] requires 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
}
```

Example 21.3.6 Consider an abstract class `Table` that implements maps from a type of keys `A` to a type of values `B`. The class has a method `set` to enter a new key / value pair into the table, and a method `get` that returns an optional value matching a given key. Finally, there is a method `apply` which is like `get`, except that it returns a given default value if the table is undefined for the given key. This class is implemented as follows.

```
abstract class Table[A, B](defaultValue: B) {
  def get(key: A): Option[B]
  def set(key: A, value: B)
  def apply(key: A) = get(key) match {
    case Some(value) => value
    case None => defaultValue
  }
}
```

Here is a concrete implementation of the `Table` class.

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

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

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

Note that `SynchronizedTable` does not pass an argument to its superclass, `Table`, even though `Table` is defined with a formal parameter. Note also that the **super** calls in `SynchronizedTable`'s `get` and `set` methods statically refer to abstract methods in class `Table`. This is legal, as long as the calling method is labeled **abstract override** (§21.2).

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

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

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

21.4 Object Definitions

Syntax:

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

An object definition defines a single object of a new class. Its most general form is **object** *m* **extends** *t*. Here, *m* is the name of the object to be defined, and *t* is a template (§21.1) of the form

sc **with** mt_1 **with** ... **with** mt_n { *stats* }

which defines the base classes, behavior and initial state of *m*. The extends clause **extends** *sc* **with** mt_1 **with** ... **with** mt_n 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 the following three definitions, which together define a class and create a single object of that class on demand:

```
final class m$cls extends t
private var m$instance = null
final def m = {
  if (m$instance == null) m$instance = new m$cls
  m$instance
}
```

Here, the **final** modifiers are omitted if the definition occurs as part of a block. The names *m*$cls and *m*$instance are inaccessible for user programs.

Note that the value 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.

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

Example 21.4.1 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*.

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

Chapter 22

Expressions

Syntax:

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

```
BlockExpr       ::=  ‘{’ CaseClauses ‘}’
                  |  ‘{’ Block ‘}’
Block           ::=  {BlockStat semi} [ResultExpr]
ResultExpr      ::=  Expr1
                  |  (Bindings | Id ‘:’ CompoundType) ‘=>’ Block
Ascription      ::=  ‘:’ CompoundType
                  |  ‘:’ Annotation {Annotation}
```

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

22.1 Literals

Syntax:

```
SimpleExpr    ::=  Literal
```

Typing of literals is as described in (§17.3); their evaluation is immediate.

A different form of literals designate classes. These are written

```
classOf[C]
```

Here, `classOf` is a method defined in `scala.Predef` (§28.5) and C is a class type. The value of such a class literal is the run-time representation of the class type C.

22.2 The *Null* Value

The **`null`** value is of type `scala.Null`, and is thus compatible with every reference type. It denotes a reference value which refers to a special “**`null`**” object. This object implements the 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.

22.3 Designators

Syntax:

```
SimpleExpr  ::=  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 (§19.1) of type T, the selection $r.x$ refers statically to a term member m of r that is identified in T by the name x.

For other expressions e, $e.x$ is typed as if it was `{ val` y `=` e`;` y`.`x `}`, for some fresh name y. The 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 undefined. The type of a designator is the type of the entity it refers to, with the following exception: The type of a path (§19.1) p which occurs as the prefix of a selection, or which has a singleton type as expected type, is the singleton type p`.type`.

The selection $e.x$ is evaluated by first evaluating the qualifier expression e, which yields an object r, say. The selection’s result is then the member r that is either defined by m or defined by a definition overriding m.

22.4 This and Super

Syntax:

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

The expression **`this`** can appear in the statement part of a template or compound type. It stands for the object being defined by the innermost template or compound type enclosing the reference. If this is a compound type, the type of **`this`** is that compound type. If it is a template of 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 innermost 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 refers statically to a member m in the least proper supertype of the innermost template containing the reference. It evaluates to the member m' in the actual supertype of that template which is equal to m or which overrides m. The statically referenced member m must be concrete, or the template containing the reference must have a member m' which overrides m and which is labeled **abstract override**.

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

The **super** prefix may be followed by a class qualifier [C], as in C.**super**[C].x. This is called a *static super reference*. In this case, the reference is to the member of x in the parent class of C whose simple name is M. That member must be uniquely defined and concrete.

Example 22.4.1 Consider the following class definitions

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

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

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

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

22.5 Function Applications

Syntax:

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

An application $f(e_1, \ldots, e_n)$ applies the function f to the argument expressions $e_1, \ldots, e_n$. If f has a method type $(T_1, \ldots, T_n)$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($e_1, \ldots, e_n$), i.e. the application of an apply method defined by f.

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

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

The case of a formal 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 expression. In other words, the evaluation order for =>-parameters is *call-by-name* whereas the evaluation order for normal parameters is *call-by-value*.

The last argument in an application may be marked as a sequence argument, e.g. e: _*. Such an argument must correspond to a repeated parameter (§20.6.2) of type S* and it must be the only argument matching this parameter (i.e. the number of formal parameters and actual arguments must be the same). Furthermore, the type of e must conform to scala.Seq[T], for some type T which conforms to S. In this case, the argument list is transformed by replacing the sequence e with its elements.

Example 22.5.1 Assume the following function which computes the sum of a variable number of arguments:

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

Then

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

both yield `10` as result. On the other hand,

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

would not typecheck.

22.6 Method Values

Syntax:

```
SimpleExpr    ::=  SimpleExpr1 ‘_’
```

The expression e `_` is well-formed if e is of method type or if e is a call-by-name parameter. If e is a method with parameters, e `_` represents e converted to a function type by eta expansion (§22.24.5). If e is a parameterless method or call-by-name parameter of type `=>`T, e `_` represents the function of type `() =>` T, which evaluates e when it is applied to the empty parameterlist `()`.

Example 22.6.1 The method values in the left column are each equivalent to the anonymous functions (§22.22) on their right.

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

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

22.7 Type Applications

Syntax:

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

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

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

Type applications can be omitted if local type inference (§22.24.4) can infer best type parameters for a polymorphic functions from the types of the actual function arguments and the expected result type.

22.8 Tuples

Syntax:

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

A tuple expression $(e_1, \ldots, e_n)$ is an alias for the class instance creation `scala.Tuple`$n(e_1, \ldots, e_n)$, where $n \geq 2$. The expression may also be written with a trailing comma, i.e. $(e_1, \ldots, e_n,)$. Unary tuples can be expressed in this syntax only by using a trailing comma, i.e. $(e,)$. Finally, the empty tuple `()` is the unique value of type `scala.Unit`.

22.9 Instance Creation Expressions

Syntax:

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

A simple instance creation expression is of the form **new** c where c is a constructor invocation (§21.1.1). 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 (§21.3). The expression is evaluated by creating a fresh object of type T which is is initialized by evaluating c. The type of the expression is T.

A general instance creation expression is of the form **new** t for some template t (§21.1). Such an expression is equivalent to the block

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

where a is a fresh name of an *anonymous class.*

22.10 Blocks

Syntax:

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

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

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

The type of a block s_1; …; 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, there are two other possibilities:

1. If a fully defined expected type is given, the type of the block is instead assumed to be the expected type.

2. Otherwise, if the type of e is an anonymous class a introduced by the expansion of an instance creation expression (§22.9), the type of the block is taken to be the least class type or refinement type which is a proper supertype of the type a.

It is a compile-time error if neither of the previous two clauses applies.

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 22.10.1 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.

22.11 Prefix, Infix, and Postfix Operations

Syntax:

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

Expressions can be constructed from operands and operators.

22.11.1 Prefix Operations

A prefix operation *op e* consists of a prefix operator *op*, which must be one of the identifiers '+', '-', '`!`' or '~'. The expression *op e* is equivalent to the postfix method application `e.unary_`*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)`.

22.11.2 Postfix Operations

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

22.11.3 Infix Operations

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

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

```
(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 in a colon '`:`' are right-associative. All other operators are left-associative.

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

- If there are several infix operations in an expression, then operators with higher precedence bind more closely than operators with lower precedence.

- If there are consecutive infix operations $e_0\ op_1\ e_1\ op_2 \ldots op_n\ e_n$ 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 $(e_1\ op_1\ e_2)\ op_2$.

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

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=e1; e2.op(x) }`, where x is a fresh name.

22.11.4 Assignment Operators

An assignment operator is an operator symbol (syntax category op in (§17.1)) that ends in an equals sign "=". Assignment operators are treated specially in that they can be expanded to assignments if no other interpretation is valid.

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

```
l = l + r
```

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

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

1. The left-hand-side l does not have a member named +=, and also cannot be converted by an implicit conversion (§22.24) to a value with a member named +=.
2. The assignment `l = l + r` is type-correct. In particular this implies that l refers to a variable or object that can be assigned to, and that is convertible to a value with a member named +.

22.12 Typed Expressions

Syntax:

```
Expr1            ::=  PostfixExpr ':' CompoundType
```

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

Example 22.12.1 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
```

22.13 Annotated Expressions

Syntax:

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

An annotated expression e`: @`a_1 … `@`a_n attaches annotations $a_1, \ldots, a_n$ to the expression e (§27).

22.14 Assignments

Syntax:

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

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

An assignment f($args$) `=` 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 22.14.1 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
  }
```

22.15 Conditional Expressions

Syntax:

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

The conditional expression **if** (e_1) e_2 **else** e_3 chooses one of the values of e_2 and e_3, depending on the value of e_1. The condition e_1 is expected to conform to type `boolean`. The then-part e_2 and the else-part e_3 are both expected to conform to the expected type of the conditional expression. The type of the conditional expression is the 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`.

22.16 While Loop Expressions

Syntax:

```
Expr1          ::=  while '(' Expr ')' {nl} Expr
```

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

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

22.17 Do Loop Expressions

Syntax:

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

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

22.18 For-Comprehensions

Syntax:

```
Expr1          ::=  for '(' Enumerators ')' {nl} [yield] Expr
                  |  for '{' Enumerators '}' {nl} [yield] Expr
```

```
Enumerators    ::=  Generator {semi Enumerator}
Enumerator     ::=  Generator
                 |  Guard
                 |  val Pattern1 ‘=’ Expr
Generator      ::=  Pattern1 ‘<-’ Expr [Guard]
Guard          ::=  ‘if’ PostfixExpr
```

A comprehension **for** ($enums$) **yield** e evaluates expression e for each binding generated by the enumerators *enums*. An enumerator sequence always starts with a generator; this can be followed by further generators, value definitions, or guards. A *generator* p `<-` e produces bindings from an expression e which is matched in some way against pattern p. A *value definition* **val** p `=` e binds the value name p (or several names in a pattern p) to the result of evaluating the expression e. A *guard* **if** e contains a boolean expression which restricts enumerated bindings. The precise meaning of generators and guards is defined by translation to invocations of four methods: `map`, `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 p `<-` e, where p is not irrefutable (§24.1) for the type of e is replaced by

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

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

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

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

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

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

- A for-comprehension

  for (p `<-` e; p' `<-` e' ...) e'' .

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

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

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

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

Example 22.18.1 The following code produces all pairs of numbers between 1 and $n-1$ whose sums are prime.

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

The for-comprehension is translated to:

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

Example 22.18.2 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 (i <- Array.range(0, xss(0).length)) yield
    Array(for (xs <- xss) yield xs(i))
```

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

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

Finally, here is a function to compute the product of two matrices. Compare with the imperative version of Example 22.14.1.

```
def matmul(xss: Array[Array[double]], yss: Array[Array[double]]) = {
  val ysst = transpose(yss)
```

```
    for (xs <- xs) yield
      for (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`.

22.19 Return Expressions

Syntax:

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

A return expression **`return`** e must occur inside the body of some enclosing named method or function. The innermost enclosing named method or function, f, must have an explicitly declared result type, and the type of e must conform to it. The return expression evaluates the expression e and returns its value as the result of f. The evaluation of any statements or expressions following the return expression is omitted. The type of a return expression is `scala.Nothing`.

If the return expression is itself part of a closure, it is possible that the enclosing instance of f has already returned before the return expression is executed. In that case, a `scala.runtime.NonLocalReturnException` is thrown.

22.20 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 (§22.21) which handles the thrown exception, evaluation resumes with the handler; otherwise the thread executing the **`throw`** is aborted. The type of a throw expression is `scala.Nothing`.

22.21 Try Expressions

Syntax:

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

A try expression is of the form **try** { b } **catch** h where the handler h is a pattern matching anonymous function (§24.5)

{ **case** p_1 => b_1 ... **case** p_n => b_n } .

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

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

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

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

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

22.22 Anonymous Functions

Syntax:

```
Expr1          ::=  (Bindings | Id) '=>' Expr
ResultExpr     ::=  (Bindings | Id [':' CompoundType]) '=>' Block
Bindings       ::=  '(' Binding {',' Binding} ')'
Binding        ::=  id [':' Type]
```

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

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

The anonymous function is evaluated as the instance creation expression

```
new scala.Functionn[T1, ..., Tn, T] {
  def apply(x1: T1, ..., xn: Tn): T = e
}
```

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

Example 22.22.1 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.
```

Implicit Anonymous Functions

Syntax:

```
SimpleExpr1  ::=  ‘_’
```

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

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

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

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

Example 22.22.2 The implicit anonymous functions in the left column are each equivalent to the anonymous functions on their right.

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

22.23 Statements

Syntax:

```
BlockStat    ::=  Import
               |  [implicit] Def
               |  {LocalModifier} TmplDef
               |  Expr1
               |
TemplateStat ::=  Import
               |  {Annotation} {Modifier} Def
               |  {Annotation} {Modifier} Dcl
               |  Expr
               |
```

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

Block statements may be definitions which bind local names in the block. The only modifiers allowed in block-local definitions are modifiers **abstract**, **final**, or **sealed** preceding a class or object definition.

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

22.24 Implicit Conversions

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

We say, a type T is *compatible* to a type U if T conforms to U after applying eta-expansion (§22.24.5) and view applications (§23.3).

22.24.1 Value Conversions

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

Overloading Resolution. If an expression denotes several possible members of a class, overloading resolution (§22.24.3) is applied to pick a unique member.

Type Instantiation. An expression e of polymorphic type

$$[a_1 \; \texttt{>:} \; L_1 \; \texttt{<:} \; U_1, \ldots, a_n \; \texttt{>:} \; L_n \; \texttt{<:} \; U_n]T$$

which does not appear as the function part of a type application is converted to a type instance of T by determining with local type inference (§22.24.4) instance types $T_1, \ldots, T_n$ for the type variables $a_1, \ldots, a_n$ and implicitly embedding e in the type application $e[T_1, \ldots, T_n]$ (§22.7).

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

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

View Application. If none of the previous conversions applies, and the e's type does not conform to the expected type *pt*, it is attempted to convert e to the expected type with a view (§23.3).

22.24.2 Method Conversions

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

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

Implicit Application. If the method takes only implicit parameters, implicit arguments are passed following the rules of §23.2.

Eta Expansion. Otherwise, if the method is not a constructor, and the expected type pt is a function type $(Ts') \Rightarrow T'$ eta-expansion (§22.24.5) is performed on the expression e.

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

22.24.3 Overloading Resolution

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

Assume first that e appears as a function in an application, as in e`(`$args$`)`. 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 an undefined 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.

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

Assume finally that e does not appear as a function in either an application or a type application. If an expected type is given, let $\mathscr{B}$ be the set of those alternatives in $\mathscr{A}$ which are compatible (§22.24) 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 $\mathscr{B}$.

In both cases, it is an error if the most specific alternative is defined in a class C, and there is another applicable alternative which is defined in a true subclass of C.

Example 22.24.1 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.

22.24.4 Local Type Inference

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

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

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

Case 2: Values. If the expression e appears as a value without being applied to value arguments, the type arguments are inferred by solving a constraint system which relates the expression's type T with the expected type *pt*. Without loss of generality we can assume that T is a value type; if it is a method type we apply eta-expansion (§22.24.5) to convert it to a function type. Solving means finding a substitution σ of types T_i for the type parameters a_i such that

- All type parameter bounds are respected, i.e. $\sigma L_i <: \sigma a_i$ and $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The expression's type conforms to the expected type, i.e. $\sigma T <: \sigma pt$.

It is a compile time error if no such substitution exists. If several substitutions exist, local-type inference will choose for each type variable a_i a minimal or maximal type T_i of the solution space. A *maximal* type T_i will be chosen if the type parameter a_i appears contravariantly (§20.5) in the type T of the expression. A *minimal* type T_i will be chosen in all other situations, i.e. if the variable appears covariantly, non-variantly or not at all in the type T. We call such a substitution an *optimal solution* of the given constraint system for the type T.

Case 3: Methods. The last case applies if the expression e appears in an application $e(d_1, \ldots, d_m)$. In that case T is a method type $(R_1, \ldots, R_m)T'$. Without loss of generality we can assume that the result type T' is a value type; if it is a method type we apply eta-expansion (§22.24.5) to convert it to a function type. One computes first the types S_j of the argument expressions d_j, using two alternative schemes. Each argument expression d_j is typed first with the expected type R_j, in which the type parameters $a_1, \ldots, a_n$ are taken as type constants. If this fails, the argument d_j is typed instead with an expected type R'_j which results from R_j by replacing every type parameter in $a_1, \ldots, a_n$ with *undefined*.

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

- All type parameter bounds are respected, i.e. $\sigma L_i <: \sigma a_i$ and $\sigma a_i <: \sigma U_i$ for $i = 1, \ldots, n$.
- The method's result type T' conforms to the expected type, i.e. $\sigma T' <: \sigma pt$.
- Each argument type conforms to the corresponding formal parameter type, i.e. $\sigma S_j <: \sigma R_j$ for $j = 1, \ldots, m$.

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

All or parts of an expected type *pt* may be undefined. The rules for conformance (§19.5.2) are extended to this case by adding the rule that for any type T the following two statements are always true:

$$\mathit{undefined} <: T \qquad \text{and} \qquad T <: \mathit{undefined}.$$

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

Example 22.24.2 Consider the two methods:

```
def cons[a](x: a, xs: List[a]): List[a] = x :: xs
def nil[b]: List[b] = Nil
```

and the definition

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

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

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

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

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

```
b = scala.Nothing .
```

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

```
int <: a?
List[scala.Nothing] <: List[a?]
List[a?] <: undefined
```

The optimal solution of this constraint system is

```
a = int ,
```

so `int` is the type inferred for a.

Example 22.24.3 Consider now the definition

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

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

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

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

```
String <: a?
List[int] <: List[a?]
List[a?] <: undefined
```

The optimal solution of this constraint system is

```
a = scala.Any ,
```

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

22.24.5 Eta Expansion

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

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

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

If the expression e has a single call-by-name parameter (i.e. it is of type `(=>`T`)`U, for some types T and U), eta-expansion of e yields a value of type `ByNameFunction`. The latter is defined as follows.

```
trait ByNameFunction[-A, +B] extends AnyRef {
  def apply(x: => A): B
  override def toString() = "<function>"
}
```

Eta expansion is not applicable to methods where a call-by-name parameter appears together with other parameters in one parameter section. Neither is it applicable to methods with repeated parameters `x: T*` (§20.6.2).

Chapter 23

Implicit Parameters and Views

23.1 The Implicit Modifier

Syntax:

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

Template members and parameters labeled with an **implicit** modifier can be passed to implicit parameters (§23.2) and can be used as implicit conversions called views (§23.3). The **implicit** modifier is illegal for all type members, as well as for top-level (§25.2) objects.

Example 23.1.1 The following code defines an abstract class of monoids and two concrete implementations, `StringMonoid` and `IntMonoid`. The two implementations are marked implicit.

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

23.2 Implicit Parameters

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

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

The actual arguments that are eligible to be passed to an implicit parameter of type T fall into two categories. First, eligible are all identifiers x that can be accessed at the point of the method call without a prefix and that denote an implicit definition (§23.1) or an implicit parameter. An eligible identifier may thus be a local name, or a member of an enclosing template, or it may be have been made accessible without a prefix through an import clause (§20.7). Second, eligible are also all **`implicit`** members of some object that belongs to the implicit scope of the implicit parameter's type, T.

The *implicit scope* of a type T consists of all companion modules (§21.4) of classes that are associated with the implicit parameter's type. Here, we say a class C is *associated* with a type T, if it is a base class (§21.1.2) of some part of T. The *parts* of a type T are:

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

If there are several eligible arguments which match the implicit parameter's type, a most specific one will be chosen using the rules of static overloading resolution (§22.24.3).

Example 23.2.1 Assuming the classes from Example 23.1.1, here is a method which computes the sum of a list of elements using the monoid's `add` and `unit` operations.

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

The monoid in question is marked as an implicit parameter, and can therefore be inferred based on the type of the list. Consider for instance the call

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

in a context where `stringMonoid` and `intMonoid` are visible. We know that the formal type parameter `a` of `sum` needs to be instantiated to `Int`. The only eligible object which matches the implicit formal parameter type `Monoid[Int]` is `intMonoid` so this object will be passed as implicit parameter.

This discussion also shows that implicit parameters are inferred after any type arguments are inferred (§22.24.4).

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

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

Assume in addition a method

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

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

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

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

```
sort(yss)
```

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

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

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

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

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

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

To prevent such infinite expansions, we require that every implicit method definition is contractive.

A method definition is *contractive* if the type of every implicit parameter type is properly contained in the type that is obtained by removing all implicit parameters from the method type and converting the rest to a function type.

A type T is *contained* in a type U if one of the following holds:

- T is the same as some part of U,
- U is a function type and T is not.
- U and T are both function types, and the arity of U is greater than the arity of T.
- U and T both parameterized types (including function types) with the same type constructor, and each type argument of T is contained in the corresponding type argument of U.

A type T is *properly contained* in a type U if T is contained in U and different from U.

Example 23.2.2 The type of `list2ordered` is

```
(List[a])(implicit a => Ordered[a]): Ordered[List[a]] .
```

This type is contractive, because the type of the implicit parameter, `a => Ordered[a]`, is properly contained in the function type of the method without implicit parameters, `List[a] => Ordered[List[a]]`.

The type of `magic` is

```
(a)(implicit a => Ordered[a]): Ordered[a] .
```

This type is not contractive, because the type of the implicit parameter, `a => Ordered[a]`, is the same as the function type of the method without implicit parameters.

23.3 Views

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

Views are applied in two situations.

1. If an expression e is of type T, and T does not conform to the expression's expected type *pt*. In this case an implicit v is searched which is applicable to e and whose result type conforms to *pt*. The search proceeds as in the case of

implicit parameters, where the implicit scope is the one of T => *pt*. If such a view is found, the expression e is converted to v(*e*).

2. In a selection $e.m$ with e of type T, if the selector m does not denote a member of T. In this case, a view v is searched which is applicable to e and whose result contains a member named m. The search proceeds as in the case of implicit parameters, where the implicit scope is the one of T. If such a view is found, the selection $e.m$ is converted to v(*e*).*m*.

As for implicit parameters, overloading resolution is applied if there are several possible candidates.

Example 23.3.1 Class `scala.Ordered[a]` contains a method

```
def <= [b >: a](that: b)(implicit b2ordered: b => Ordered[b]): boolean .
```

Assume two lists `xs` and `ys` of type `List[int]` and assume that the `list2ordered` and `int2ordered` methods defined in §23.2 are in scope. Then the operation

```
xs <= ys
```

is legal, and is expanded to:

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

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

23.4 View Bounds

Syntax:

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

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

A method or class containing such a type parameter is treated as being equivalent to a method with a view parameter. E.g.

def f[a <% T](ps): R = ...

is expanded to

```
def f[a](ps)(implicit v: a => T): R = ...
```

where v is a fresh name for the implicit parameter. Since traits do not take constructor parameters, this translation does not work for them. Consequently, type-parameters in traits may not be view-bounded.

Example 23.4.1 The <= method mentioned in Example 23.3.1 can be declared more concisely as follows:

```
def <= [b >: a <% Ordered[b]](that: b): boolean
```

Chapter 24

Pattern Matching

24.1 Patterns

Syntax:

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

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

Example 24.1.1 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 `Some(x)` matches values of the form `Some(`v`)`, binding x to the argument value v of the `Some` constructor.
3. 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.
4. The pattern `x :: y :: xs` matches lists of length ≥ 2, binding x to the list's first element, y to the list's second element, and xs to the remainder.
5. The pattern `1 | 2 | 3` matches the integers between 1 and 3.

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

24.1.1 Variable Patterns

Syntax:

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

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

24.1.2 Typed Patterns

Syntax:

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

A typed pattern $x : T$ consists of a pattern variable x and a type pattern T. This pattern matches any value matched by the type pattern T (§24.2); it binds the variable name to that value.

24.1.3 Literal Patterns

Syntax:

```
SimplePattern   ::=  Literal
```

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

24.1.4 Stable Identifier Patterns

Syntax:

```
SimplePattern   ::=  StableId
```

A stable identifier pattern is a stable identifier r (§19.1). The type of r must conform to the expected type of the pattern. The pattern matches any value v such that r == v (§28.1).

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

Example 24.1.2 Consider the following function definition:

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

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

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

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

24.1.5 Constructor Patterns

Syntax:

```
SimplePattern   ::=  StableId ‘(’ [Patterns [‘,’]] ‘)
```

A constructor pattern is of the form $c(p_1, \ldots, p_n)$ where $n \geq 0$. It consists of a stable identifier c, followed by element patterns $p_1, \ldots, p_n$. The constructor c is a simple or qualified name which denotes a case class (§21.3.2). If the case class is monomorphic, then it must conform to the expected type of the pattern, and the formal parameter types of x's primary constructor (§21.3) are taken as the expected types of the element patterns $p_1, \ldots, p_n$. If the case class is polymorphic, then its type parameters are instantiated so that the instantiation of c conforms to the expected type of the pattern. The instantiated formal parameter types of c's primary constructor are then taken as the expected types of the component patterns $p_1, \ldots, p_n$. The pattern matches all objects created from constructor invocations $c(v_1, \ldots, v_n)$ where each element pattern p_i matches the corresponding value v_i.

A special case arises when c's formal parameter types end in a repeated parameter. This is further discussed in (§24.1.8).

24.1.6 Tuple Patterns

Syntax:

```
SimplePattern   ::=  ‘(’ [Patterns [‘,’]] ‘)’
```

A tuple pattern $(p_1, \ldots, p_n)$ is an alias for the constructor pattern `scala.Tuple`$n(p_1, \ldots, p_n)$, where $n \geq 2$. The pattern may also be written with a trailing comma, i.e. $(p_1, \ldots, p_n,)$. Unary tuple patterns can be expressed in this syntax only by using a trailing comma, i.e. $(p,)$. Finally, the empty tuple `()` is the unique value of type `scala.Unit`.

24.1.7 Extractor Patterns

Syntax:

```
SimplePattern   ::=  StableId ‘(’ [Patterns [‘,’]] ‘)’
```

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

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

$n = 0$ and `unapply`'s result type is `boolean`. In this case the extractor pattern matches all values v for which x`.unapply(`v`)` yields **`true`**.

$n = 1$ and `unapply`'s result type is `Option[`T`]`, for some type T. In this case, the (only) argument pattern p_1 is typed in turn with expected type T. The extractor pattern matches then all values v for which x`.unapply(`v`)` yields a value of form `Some(`v_1`)`, and p_1 matches v_1.

$n > 1$ and `unapply`'s result type is `Option[{`$T_1, \ldots, T_n$`}]`, for some types $T_1, \ldots, T_n$. In this case, the argument patterns $p_1, \ldots, p_n$ are typed in turn with expected types $T_1, \ldots, T_n$. The extractor pattern matches then all values v for which x`.unapply(`v`)` yields a value of form `Some({`$v_1, \ldots, v_n$`})`, and each pattern p_i matches the corresponding value v_i.

An `unapplySeq` method in an object x matches the pattern $x(p_1, \ldots, p_n)$ if it takes exactly one argument and its result type is of the form `Option[`S`]`, where S is a subtype of `Seq[`T`]` for some element type T. This case is further discussed in (§24.1.8).

24.1.8 Pattern Sequences

Syntax:

```
SimplePattern ::= StableId ‘(’ [Patterns ‘,’] ‘_’ ‘*’ ‘)’
```

A pattern sequence $p_1, \ldots, p_n$ appears in two contexts. First, in a constructor pattern $c(q_1, \ldots, q_m, p_1, \ldots, p_n)$, where c is a case class which has $m+1$ primary constructor parameters, ending in a repeated parameter (§20.6.2) of type $S*$. Second, in an extractor pattern $x(p_1, \ldots, p_n)$ if the extractor object x has an `unapplySeq` method with a result type conforming to `Seq[`S`]`, but does not have an `unapply` method that matches $p_1, \ldots, p_n$. The expected type for the pattern sequence is in each case the type S.

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

24.1.9 Infix Operation Patterns

Syntax:

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

An infix operation pattern $p\ op\ q$ is a shorthand for the constructor or extractor pattern $op(p, q)$. The precedence and associativity of operators in patterns is the same as in expressions (§22.11).

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

24.1.10 Pattern Alternatives

Syntax:

```
Pattern   ::=  Pattern1 { ‘|’ Pattern1 }
```

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

24.1.11 XML Patterns

XML patterns are treated in §26.2.

24.1.12 Regular Expression Patterns

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

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

24.1.13 Irrefutable Patterns

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

1. p is a variable pattern,
2. p is a typed pattern $x : T'$, and $T <: T'$,
3. p is a constructor pattern $c(p_1, \ldots, p_n)$, the type T is an instance of class c, the primary constructor (§21.3) of type T has argument types $T_1, \ldots, T_n$, and each p_i is irrefutable for T_i.

24.2 Type Patterns

Syntax:

```
TypePat          ::=  CompoundTypePat {id [nl] CompoundTypePat}
CompoundTypePat  ::=  AnnotTypePat {with AnnotTypePat}
AnnotTypePat     ::=  {Annotation} SimpleTypePat
SimpleTypePat    ::=  SimpleTypePat1 [TypePatArgs]
SimpleTypePat1   ::=  SimpleTypePat1 ‘#’ id
                   |  StableId
                   |  Path ‘.’ type
                   |  ‘(’ ArgTypePats [‘,’] ‘)’

TypePatArgs      ::=  ‘[’ ArgTypePats ’]’
ArgTypePats      ::=  ArgTypePat {‘,’ ArgTypePat}
ArgTypePat       ::=  varid
                   |  ‘_’
                   |  Type
```

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

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

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

- A singleton type $p.type$. This type pattern matches only the value denoted by the path p (that is, a pattern match involved a comparison of the matched value with p using method `eq` in class `AnyRef`).

- A compound type pattern T_1 **`with`** $\ldots$ **`with`** T_n where each T_i is a type pattern. This type pattern matches all values that are matched by each of the type patterns T_i.

- A parameterized type pattern $T[a_1, \ldots, a_n]$, where the a_i are type variable patterns or wildcards _. This type pattern matches all values which match T for some arbitrary instantiation of the type variables and wildcards. The bounds or alias type of these type variable are determined as described in (§24.3).

- A parameterized type pattern `scala.Array`$[T_1]$, where T_1 is a type pattern. This type pattern matches any non-null instance of type `scala.Array`$[U_1]$, where U_1 is a type matched by T_1.

Also accepted is a parameterized type pattern of the form $T[U_1, \ldots, U_n]$ where T is different from `scala.Array` and some of the U_i are types instead of type variable patterns or wildcards. However, such a type pattern will be translated to the erasure (§19.6) of $T[U_1, \ldots, U_n]$. The Scala compiler will issue an "unchecked" warning for these patterns to flag the possible loss of type-safety.

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

24.3 Type Parameter Inference in Patterns

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

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

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

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

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

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

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

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

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

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

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

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

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

Example 24.3.1 Consider the program fragment:

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

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

On the other hand, if x is declared as

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

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

Example 24.3.2 Consider the program fragment:

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

Scala does not maintain information about type arguments at run-time, so there is no way to check that x is a list of strings. Instead, the Scala compiler will erase (§19.6) the pattern to `List[_]`; that is, it will only test whether the top-level runtime-class of the value `x` conforms to `List`, and the pattern match will succeed if it does. This might lead to a class cast exception later on, in the case where the list `x` contains elements other than strings. The Scala compiler will flag this potential loss of type-safety with an “unchecked” warning message.

Example 24.3.3 Consider the program fragment

```
class Term[a]
class Number(val n: int) extends Term[int]
def f[b](t: Term[b]): b = t match {
  case y: Number => y.n
}
```

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

24.4 Pattern Matching Expressions

Syntax:

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

A pattern matching expression

e **match** { **case** p_1 => b_1 ... **case** p_n => b_n }

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

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

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

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

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

When applying a pattern matching expression to a selector value, patterns are tried in sequence until one is found which matches the selector value (§24.1). 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.

If the selector of a pattern match is an instance of a **`sealed`** class (§21.2), the compilation of pattern matching can emit warnings which diagnose that a given set of patterns is not exhaustive, i.e. that there is a possibility of a `MatchError` being raised at run-time.

Example 24.4.1 Consider the following definitions of arithmetic terms:

```
abstract class Term[T]
case class Lit(x: int) extends Term[int]
case class Succ(t: Term[int]) extends Term[int]
case class IsZero(t: Term[int]) extends Term[boolean]
case class If[T](c: Term[boolean],
                 t1: Term[T],
                 t2: Term[T]) extends Term[T]
```

There are terms to represent numeric literals, incrementation, a zero test, and a conditional. Every term carries as a type parameter the type of the expression it representes (either `int` or `boolean`).

A type-safe evaluator for such terms can be written as follows.

```
def eval[T](t: Term[T]): T = t match {
  case Lit(n)        => n
  case Succ(u)       => eval(u) + 1
  case IsZero(u)     => eval(u) == 0
  case If(c, u1, u2) => eval(if (eval(c)) u1 else u2)
}
```

Note that the evaluator makes crucial use of the fact that type parameters of enclosing methods can acquire new bounds through pattern matching.

For instance, the type of the pattern in the second case, `Succ(u)`, is `int`. It conforms to the selector type `T` only if we assume an upper and lower bound of `int` for `T`. Under the assumption `int <: T <: int` we can also verify that the type right hand side of the second case, `int` conforms to its expected type, `T`.

24.5 Pattern Matching Anonymous Functions

Syntax:

```
BlockExpr ::= ‘{’ CaseClauses ‘}’
```

An anonymous function can be defined by a sequence of cases

```
{ case p1 => b1 ... case pn => bn }
```

which appear as an expression without a prior **match**. The expected type of such an expression must in part be defined. It must be either `scala.Function`k`[`$S_1, \ldots, S_k$`, `R`]` for some $k > 0$, or `scala.PartialFunction[`S_1`, `R`]`, where the argument type(s) $S_1, \ldots, S_k$ must be fully determined, but the result type R may be undetermined.

If the expected type is `scala.Function`k`[`$S_1, \ldots, S_k$`, `R`]`, the expression is taken to be equivalent to the anonymous function:

```
(x1:S1,...,xk:Sk) => (x1,...,xk) match {
  case p1 => b1 ... case pn => bn
}
```

Here, each x_i is a fresh name. As was shown in (§22.22), this anonymous function is in turn equivalent to the following instance creation expression, where T is the least upper bound of the types of all b_i.

```
new scala.Functionk[S1,...,Sk, T] {
  def apply(x1:S1,...,xk:Sk): T = (x1,...,xk) match {
    case p1 => b1 ... case pn => bn
  }
}
```

If the expected type is `scala.PartialFunction[`S`, `R`]`, the expression is taken to be equivalent to the following instance creation expression:

```
new scala.PartialFunction[S, T] {
  def apply(x: S): T = x match {
    case p1 => b1 ... case pn => bn
  }
  def isDefinedAt(x: S): boolean = {
    case p1 => true ... case pn => true
```

```
      case _ => false
    }
  }
```

Here, x is a fresh name and T is the least upper bound of the types of all b_i. The final default case in the isDefinedAt method is omitted if one of the patterns $p_1, \ldots, p_n$ is already a variable or wildcard pattern.

Example 24.5.1 Here is a method which uses a fold-left operation /: to compute the scalar product of two vectors:

```
  def scalarProduct(xs: Array[Double], ys: Array[Double]) =
    (0.0 /: (xs zip ys)) {
      case (a, (b, c)) => a + b * c
    }
```

The case clauses in this code are equivalent to the following anonymous funciton:

```
    (x, y) => (x, y) match {
      case (a, (b, c)) => a + b * c
    }
```

Chapter 25

Top-Level Definitions

25.1 Compilation Units

Syntax:

```
CompilationUnit  ::=  [package QualId semi] TopStatSeq
TopStatSeq       ::=  TopStat {semi TopStat}
TopStat          ::=  {Annotation} {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` (§28.5). Members of a later import in that order hide members of an earlier import.

25.2 Packagings

Syntax:

```
Packaging       ::=  package QualId [nl] ‘{’ TopStatSeq ‘}’
```

A package is a special object which defines a set of member classes, objects and packages. Unlike other objects, packages are not introduced by a definition. Instead, the set of members of a package is determined by packagings.

A packaging **package** *p* { *ds* } injects all definitions in *ds* as members into the package whose qualified name is *p*. Members of a package are called *top-level* definitions. If a definition in *ds* is labeled **private**, it is visible only for other members in the package.

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.

25.3 Package References

Syntax:

```
QualId          ::=  id {'.' id}
```

A reference to a package takes the form of a qualified identifier. Like all other references, package references are relative. That is, a package reference starting in a name *p* will be looked up in the closest enclosing scope that defines a member named *p*.

The special predefined name `_root_` refers to the outermost root package which contains all top-level packages.

Example 25.3.1 Consider the following program:

```
package b {
  class B
}

package a.b {
  class A {
    val x = new _root_.b.B
  }
}
```

Here, the reference `_root_.b.B` refers to class `B` in the toplevel package `b`. If the `_root_` prefix had been omitted, the name `b` would instead resolve to the package `a.b`, and, provided that package does not also contain a class `B`, a compiler-time error would result.

25.4 Programs

A *program* is a top-level object that has a member method `main` of type `(Array[String])unit`. Programs can be executed from a command shell. The program's command arguments are are passed to the `main` method as a parameter of type `Array[String]`.

The `main` method of a program can be directly defined in the object, or it can be inherited. The scala library defines a class `scala.Application` that defines an empty inherited `main` method. An objects m inheriting from this class is thus a program, which executes the initializaton code of the object m.

Example 25.4.1 The following example will create a hello world program by defining a method `main` in module `test.HelloWorld`.

```
package test

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

This program can be started by the command

```
scala test.HelloWorld
```

In a Java environment, the command

```
java test.HelloWorld
```

would work as well.

`HelloWorld` can also be defined without a `main` method by inheriting from `Application` instead:

```
package test
object HelloWord extends Application {
  System.out.println("hello world")
}
```

Chapter 26

XML expressions and patterns

By Burak Emir

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.

26.1 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 (§17.5).

Syntax:

```
XmlExpr ::= XmlContent {Element}
```

Well-formedness constraints of the XML specification apply, which means for instance that start tags and end tags must match, and attributes may only be defined once, with the exception of constraints related to entity resolution.

The following productions describe Scala's extensible markup language, designed as close as possible to the W3C extensible markup language standard. Only the productions for attribute values and character data are changed. Scala does not support 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      ::=    XmlContent
                |    Reference
                |    ScalaExpr
XmlContent    ::=    Element
                |    CDSect
                |    PI
                |    Comment
```

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 behavior 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  ‘:’
```

26.2 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 (§17.5).

Syntax:

```
XmlPattern  ::= ElementPattern
```

Well-formedness constraints of the XML specification apply.

An XML pattern has to be a single element pattern. It matches exactly those runtime representations of an XML tree that have the same structure as described by the pattern. XML patterns may contain Scala patterns(§24.4).

Whitespace is treated the same way as in XML expressions. Patterns that are entity references, CDATA sections, processing instructions and comments match runtime representations which are the the same.

By default, beginning and trailing whitespace in element content is removed, and consecutive occurrences of whitespace are replaced by a single space character \u0020. This behavior can be changed to preserve all whitespace with a compiler option.

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 27

User-Defined Annotations

Syntax:

```
Annotation      ::=  ‘@’ AnnotationExpr [nl]
AnnotationExpr  ::=  Constr [‘{’ {NameValuePair} ‘}’]
NameValuePair   ::=  val id ‘=’ PrefixExpr
```

User-defined annotations associate meta-information with definitions. A simple annotation has the form `@`c or `@`$c(a_1, \ldots, a_n)$. Here, c is a constructor of a class C, which must conform to the class `scala.Annotation`. All given constructor arguments $a_1, \ldots, a_n$ must be constant expressions. The constructor may be optionally followed by a list of name/value pairs in braces, e.g. $\{n_1 = c_1, \ldots, n_k = c_k\}$. All values c_i in that list must be constant expressions.

Annotations may apply to definitions or declarations, types, or expressions. An annotation of a definition or declaration appears in front of that definition. An annotation of a type appears in front of that type. An annotation of an expression e appears after the expression e, separated by a colon. More than one annotation clause may apply to an entity. The order in which these annotations are given does not matter.

Examples:

```
@serializable class C { ... }        // A class annotation.
@transient @volatile var m: int      // A variable annotation
@local String                        // A type annotation
(e: @unchecked) match { ... }        // An expression annotation
```

The meaning of annotation clauses is implementation-dependent. On the Java platform, the following annotations have a standard meaning.

`@transient`

Marks a field to be non-persistent; this is equivalent to the `transient` modifier in Java.

`@volatile`

Marks a field which can change its value outside the control of the program; this is equivalent to the `volatile` modifier in Java.

`@serializable`

Marks a class to be serializable; this is equivalent to inheriting from the `java.io.Serializable` interface in Java.

`@SerialVersionUID(<longlit>)`

Attaches a serial version identifier (a `long` constant) to a class. This is equivalent to a the following field definition in Java:

```
private final static SerialVersionUID = <longlit>
```

`@throws(<classlit>)`

A Java compiler checks that a program contains handlers for checked exceptions by analyzing which checked exceptions can result from execution of a method or constructor. For each checked exception which is a possible result, the `throws` clause for the method or constructor must mention the class of that exception or one of the superclasses of the class of that exception. Since Scala has no checked exceptions, Scala methods must be annotated with one or more `throws` annotations such that Java code can catch exceptions thrown by a Scala method.

`@deprecated`

Marks a definition as deprecated. Accesses to the defined entity will then cause a deprecated warnig to be issued from the compiler. Deprecated warnings are suppressed in code that belongs itself to a definition that is labeled deprecated.

`@scala.reflect.BeanProperty`

When prefixed to a definition of some variable `X`, this annotation causes getter and setter methods `getX`, `setX` in the Java bean style to be added in the class containing the variable. The first letter of the variable appears capitalized after the `get` or `set`. When the annotation is added to the definition of an immutable value definition `X`, only a getter is generated. The construction of these methods is part of code-generation; therefore, these methods become visible only once a classfile for the containing class is generated.

`@unchecked`

When applied to the selector of a **`match`** expression, this attribute suppresses any warnings about non-exhaustive pattern matches which would otherwise be emitted. For instance, no warnings would be produced for the method definition below.

```
def f(x: Option[int]) = (x: @unchecked) match {
  case Some(y) => y
}
```

Without the `@unchecked` annotation, a Scala compiler could infer that the pattern match is non-exhaustive, and could produce a warning because `Option` is a **`sealed`** class.

Other annotations may be interpreted by platform- or application-dependent tools. Class `scala.Annotation` has two sub-traits which are used to indicate how these annotations are retained. Instances of an annotation class inheriting from trait `scala.ClassfileAnnotation` will be stored in the generated class files. Instances of an annotation class inheriting from trait `scala.StaticAnnotation` will be visible to the Scala type-checker in every compilation unit where the annotationd symbol is accessed. An annotation class can inherit from both `scala.ClassfileAnnotation` and `scala.StaticAnnotation`. If an annotation class inherits from neither `scala.ClassfileAnnotation` nor `scala.StaticAnnotation`, its instances are visible only locally during the compilation run that analyzes them.

Classes inheriting from `scala.ClassfileAnnotation` may be subject to further restrictions in order to assure that they can be mapped to the host environment. In particular, on both the Java and the .NET platforms, such classes must be toplevel; i.e. they may not be contained in another class or object.

Chapter 28

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.

28.1 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 signatures of these root classes are described by the following definitions.

```
package scala
/** The universal root class */
abstract class Any {

  /** Defined equality; abstract here */
  def equals(that: Any): boolean
```

```
    /** Semantic equality between values 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; abstract here */
    def hashCode(): Int = ...

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

    /** Type test; needs to be inlined to work as given */
    def isInstanceOf[a]: Boolean = this match {
      case x: a => true
      case _ => false
    }

    /** Type cast; needs to be inlined to work as given */ */
    def asInstanceOf[a]: a = this match {
      case x: a => x
      case _ => if (this eq null) this
                else throw new ClassCastException()
    }
  }

  /** The root class of all value types */
  final class AnyVal extends Any

  /** The root class of all reference types */
  class AnyRef extends Any {
    def equals(that: Any): Boolean     = this eq that
    final def eq(that: AnyRef): Boolean   = ...  // reference equality

    def hashCode(): Int = ...     // hashCode computed from allocation address
    def toString(): String  = ... // toString computed from hashCode and class name

  /** A mixin class for every user-defined Scala class */
  trait ScalaObject extends AnyRef
```

The test *x*.asInstanceOf[*T*] is treated specially if *T* is a numeric value type (§28.2. In this case the cast will be translated to an application of a conversion method x.to*T* (§28.2.1). For non-numeric values *x* the operation will raise a ClassCastException.

28.2 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.

28.2.1 Numeric Value Types

Classes `Double`, `Float`, `Long`, `Int`, `Char`, `Short`, and `Byte` are together called *numeric value types*. Classes `Byte`, `Short`, or `Char` are called *subrange types*. Subrange types, as well as `Int` and `Long` are called *integer types*, whereas `Float` and `Double` are called *floating point types*.

Numeric value types are ranked in the following partial order:

```
Byte - Short
             \
               Int - Long - Float - Double
             /
        Char
```

`Byte` and `Short` are the lowest-ranked types in this order, whereas `Double` is the highest-ranked. Ranking does *not* imply a conformance (§19.5.2) relationship; for instance `Int` is not a subtype of `Long`. However, object `Predef` (§28.5) defines views (§23.3) from every numeric value type to all higher-ranked numeric value types. Therefore, lower-ranked types are implicitly converted to higher-ranked types when required by the context (§22.24).

Given two numeric value types S and T, the *operation type* of S and T is defined as follows: If both S and T are subrange types then the operation type of S and T is `Int`. Otherwise the operation type of S and T is the larger of the two types wrt ranking. Given two numeric values v and w the operation type of v and w is the operation type of their run-time types.

Any numeric value type T supports the following methods.

- Comparison methods for equals (`==`), not-equals (`!=`), less-than (`<`), greater-than (`>`), less-than-or-equals (`<=`), greater-than-or-equals (`>=`), which each exist in 7 overloaded alternatives. Each alternative takes a parameter of some numeric value type. Its result type is type `Boolean`. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given comparison operation of that type.
- Arithmetic methods addition (`+`), subtraction (`-`), multiplication (`*`), division (`/`), and remainder (`%`), which each exist in 7 overloaded alternatives. Each alternative takes a parameter of some numeric value type U. Its result type is

the operation type of T and U. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given arithmetic operation of that type.

- Parameterless arithmethic methods identity (`+`) and negation (`-`), with result type T. The first of these returns the receiver unchanged, whereas the second returns its negation.
- Conversion methods `toByte`, `toShort`, `toChar`, `toInt`, `toLong`, `toFloat`, `toDouble` which convert the receiver object to the target type, using the rules of Java's numeric type cast operation. The conversion might truncate the numeric value (as when going from `Long` to `Int` or from `Int` to `Byte`) or it might lose precision (as when going from `Double` to `Float` or when converting between `Long` and `Float`).

Integer numeric value types support in addition the following operations:

- Bit manipulation methods bitewise-and (`&`), bitwise-or `|`, and bitwise-exclsuive-or (`^`), which each exist in 5 overloaded alternatives. Each alternative takes a parameter of some integer numeric value type. Its result type is the operation type of T and U. The operation is evaluated by converting the receiver and its argument to their operation type and performing the given bitwise operation of that type.
- A parameterless bit-negation method (`~`). Its result type is the reciver type T or `Int`, whichevery is larger. The operation is evaluated by converting the receiver to the result type and negating every bit in its value.
- Bit-shift methods left-shift (`<<`), arithmetic right-shift (`>>`), and unsigned right-shift (`>>>`). Each of these methods of has two overloaded alternatives, which take a parameter n of type `Int`, respectively `Long`. The result type of the operation is the reciver type T, or `Int`, whichever is larger. The operation is evaluated by converting the receiver to the result type and performing the specified shift by n bits.

Numeric value types also implement operations `equals`, `hashCode`, and `toString` from class `Any`.

The `equals` method tests whether the argument is a numeric value type. If this is true, it will perform the `==` operation which is appropriate for that type. That is, the `equals` method of a numeric value type can be thought of being defined as follows:

```
def equals(other: Any): Boolean = other match {
  case that: Byte   => this == that
  case that: Short  => this == that
  case that: Char   => this == that
  case that: Int    => this == that
  case that: Long   => this == that
```

```
    case that: Float  => this == that
    case that: Double => this == that
    case _ => false
  }
```

The hashCode method returns an integer hashcode that maps equal numeric values to equal results. It is guaranteed to be the identity for for type Int and for all subrange types.

The toString method displays its receiver as an integer or floating point number.

Example 28.2.1 As an example, here is the signature of the numeric value type Int:

```
package scala
abstract sealed class Int extends AnyVal {
  def == (that: Double): Boolean  // double equality
  def == (that: Float): Boolean   // float equality
  def == (that: Long): Boolean    // long equality
  def == (that: Int): Boolean     // int equality
  def == (that: Short): Boolean   // int equality
  def == (that: Byte): Boolean    // int equality
  def == (that: Char): Boolean    // int equality
  /* analogous for !=, <, >, <=, >= */

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

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

  def << (cnt: Int): Int          // int left shift
  def << (cnt: Long): Int         // long left shift
  /* analogous for >>, >>> */

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

```
    def toByte: Byte               // convert to Byte
    def toShort: Short             // convert to Short
    def toChar: Char               // convert to Char
    def toInt: Int                 // convert to Int
    def toLong: Long               // convert to Long
    def toFloat: Float             // convert to Float
    def toDouble: Double           // convert to Double
  }
```

28.2.2 Class Boolean

Class Boolean has only two values: **true** and **false**. It implements operations as given in the following signature:

```
package scala
abstract sealed class Boolean extends AnyVal {
  def && (p: => Boolean): Boolean  // boolean and
  def || (p: => 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 unary_!: Boolean             // boolean negation
}
```

The class also implements operations equals, hashCode, and toString from class Any.

The equals method returns **true** if the argument is the same boolean value as the receiver, **false** otherwise. The hashCode method returns 1 when invoked on **true**, and 0 when invokend on **false**. The toString method returns the receiver converted to a string, i.e. either "**true**" or "**false**".

28.2.3 Class Unit

Class Unit has only one value: (). It implements only the three methods equals, hashCode, and toString from class Any.

The equals method returns **true** if the argument is the unit value {}, **false** otherwise. The hashCode method returns a fixed, implementation-specific hash-code, The toString method returns "()".

28.3 Standard Reference Classes

This section presents some standard Scala reference classes which are treated in a special way in Scala compiler – either Scala provides syntactic sugar for them, or the Scala compiler generates special code for their operations. Other classes in the standard Scala library are documented in the Scala library documentation by HTML pages.

28.3.1 Class `String`

Scala's `String` class is usually derived from the standard String class of the underlying host system (and may be identified with it). For Scala clients the class is taken to support in each case a method

```
def + (that: Any): String
```

which concatenates its left operand with the textual representation of its right operand.

28.3.2 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 (§28.5) defines the names `Pair` as an alias of `Tuple2` and `Triple` as an alias for `Tuple3`.

28.3.3 The `Function` Classes

Scala defines function classes `Function`*n* for $n = 1, \ldots, 9$. These are defined as follows.

```
package scala
trait 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 (§28.5) defines the name `Function` as an alias of `Function1`.

28.3.4 Class `Array`

The class of generic arrays is given as follows.

```
final class Array[A](len: Int) extends Seq[A] {
  def length: Int = len
  def apply(i: Int): A = ...
  def update(i: Int, x: A): Unit = ...
  def elements: Iterator[A] = ...
  def subArray(from: Int, end: Int): Array[a] = ...
  def filter(p: a => Boolean): Array[a] = ...
  def map[b](f: a => b): Array[b] = ...
  def flatMap[b](f: a => Array[b]): Array[b] = ...
}
```

If T is not a type parameter or abstract type, the type Array[T] is represented as the native array type `[]`T in the underlying host system. In that case `length` returns the length of the array, `apply` means subscribting, and `update` means element update. Because of the syntactic sugar for `apply` and `update` operations (§22.24, we have the following correspondences between Scala and Java/C# code for operations on an array xs:

```
Scala            Java/C#
  xs.length        xs.length
  xs(i)            xs[i]
  xs(i) = e        xs[i] = e
```

Arrays also implement the sequence trait `scala.Seq` by defining an `elements` method which returns all elements of the array in an `Iterator`.

Because of the tension between parametrized types in Scala and the ad-hoc implementation of arrays in the host-languages, some subtle points need to be taken into account when dealing with arrays. These are explained in the following.

First, unlike arrays in Java or C#, arrays in Scala are *not* co-variant; That is, $S <: T$ does not imply `Array[`S`]` $<:$ `Array[`T`]` in Scala. However, it is possible to cast an array of S to an array of T if such a cast is permitted in the host enironment.

For instance `Array[String]` does not conform to `Array[Object]`, even though `String` conforms to `Object`. However, it is possible to cast an expression of type `Array[String]` to `Array[Object]`, and this cast will succeed withiout raising a

ClassCastException. Example:

```
val xs = new Array[String](2)
// val ys: Array[Object] = xs   // **** error: incompatible types
val ys: Array[Object] = xs.asInstanceOf[Array[Object]] // OK
```

Second, for *polymorphic arrays*, that have a type parameter or abstract type T as their element type, a representation different from []T might be used. However, it is guaranteed that isInstanceOf and asInstanceOf still work as if the array used the standard representation of monomorphic arrays:

```
val ss = new Array[String](2)

def f[T](xs: Array[T]): Array[String] =
  if (xs.isInstanceOf[Array[String]]) xs.asInstanceOf[Array[String])
  else throw new Error("not an instance")

f(ss)                                     // returns ss
```

The representatuon chosen for polymorphic arrays also guarantees that polymorphic array creations work as expected. An example is the following implementation of method mkArray, which creates an array of an arbitrary type T, given a sequence of T's which defines its elements.

```
def mkArray[T](elems: Seq[T]): Array[T] = {
  val result = new Array[T](elems.length)
  var i = 0
  for (elem <- elems) {
    result(i) = elem
    i = i + 1
  }
}
```

Note that under Java's erasure model of arrays the method above would not work as expected – in fact it would always return an array of Object.

Third, in a Java environment there is a method System.arraycopy which takes two objects as parameters together with start indices and a length argument, and copies elements from one object to the other, provided the objects are arrays of compatible element types. System.arraycopy will not work for Scala's polymorphic arrays because of their different representation. One should instead use method Array.copy, defined as follows:

```
package scala
object Array {
  def copy(src: AnyRef, srcPos: Int,
           dest: AnyRef, destPos: Int,
            length: Int): Unit = ...
```

Example 28.3.1 The following method duplicates a given argument array and returns a pair consisting of the original and the duplicate:

```
def duplicate[T](xs: Array[T]) = {
  val ys = new Array[T](xs.length)
  Array.copy(xs, 0, ys, 0, xs.length)
  (xs, ys)
}
```

28.4 Class Node

```
package scala.xml

trait Node {

  /** the label of this node */
  def label: String

  /** attribute axis */
  def attribute: Map[String, String]

  /** child axis (all children of this node) */
  def child: Seq[Node]

  /** descendant axis (all descendants of this node) */
  def descendant: Seq[Node] = child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  /** descendant axis (all descendants of this node) */
  def descendant_or_self: Seq[Node] = this::child.toList.flatMap {
    x => x::x.descendant.asInstanceOf[List[Node]]
  }

  override def equals(x: Any): boolean = x match {
    case that:Node =>
      that.label == this.label &&
        that.attribute.sameElements(this.attribute) &&
          that.child.sameElements(this.child)
    case _ => false
  }
  /** XPath style projection function. Returns all children of this node
   *  that are labeled with 'that. The document order is preserved.
   */
    def \(that: Symbol): NodeSeq = {
      new NodeSeq({
        that.name match {
          case "_" => child.toList
          case _ =>
            var res:List[Node] = Nil
            for (x <- child.elements if x.label == that.name) {
              res = x::res
            }
            res.reverse
        }
      })
    }

  /** XPath style projection function. Returns all nodes labeled with the
   *  name 'that from the descendant_or_self axis. Document order is preserved.
   */
   def \\(that: Symbol): NodeSeq = {
     new NodeSeq(
       that.name match {
         case "_" => this.descendant_or_self
         case _ => this.descendant_or_self.asInstanceOf[List[Node]].
         filter(x => x.label == that.name)
       })
   }

   /** hashcode for this XML node */
   override def hashCode() =
     Utility.hashCode(label, attribute.toList.hashCode(), child)

   /** string representation of this node */
   override def toString() = Utility.toXML(this)

}
```

28.5 The `Predef` Object

The `Predef` object defines standard functions and type aliases for Scala programs. It is always implicitly imported, so that all its defined members are available without qualification. Its definition for the JVM environment conforms to the following signature:

```
package scala
object Predef {

  // classOf ---------------------------------------------------------

  /** Return the runtime representation of a class type. */
  def classOf[T]: Class = null  // this is a dummy, classOf is handled by compiler.

  // Standard type aliases ---------------------------------------------

  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

  type Pair[+p, +q] = Tuple2[p, q]
  type Triple[+a, +b, +c] = Tuple3[a, b, c]

  type Function[-a,+b] = Function1[a,b]

  // Factory methods -------------------------------------------------------

  def Pair[a, b](x: a, y: b) = Tuple2(x, y)
  def Triple[a, b, c](x: a, y: b, z: c) = Tuple3(x, y, z)

  def Tuple[a1, a2](x1: a1, x2: a2) = Tuple2(x1, x2)
  def Tuple[a1, a2, a3](x1: a1, x2: a2, x3: a3) = Tuple3(x1, x2, x3)

  // analogous for tuples of length 4-9:
  ...
```

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

// analogous to above:
def Array(xs: boolean*): Array[boolean] = ...
def Array(xs: byte*)   : Array[byte]    = ...
def Array(xs: short*)  : Array[short]   = ...
def Array(xs: char*)   : Array[char]    = ...
def Array(xs: int*)    : Array[int]     = ...
def Array(xs: long*)   : Array[long]    = ...
def Array(xs: float*)  : Array[float]   = ...
def Array(xs: double*) : Array[double]  = ...
def Array(xs: unit*)   : Array[unit]    = ...

// The ‘‘catch-all’’ view ---------------------------------------

implicit def identity[a](x: a): a = x

// Views into class Ordered

implicit def int2ordered(x: int): Ordered[int] = new Ordered[int] with Proxy {
  def self: Any = x
  def compare [b >: int <% Ordered[b]](y: b): int = y match {
    case y1: int =>
      if (x < y1) -1
      else if (x > y1) 1
      else 0
    case _ => -(y compare x)
  }
}

  // The implementations of following methods are analogous to the last one:

implicit def char2ordered(x: char): Ordered[char] = ...
implicit def long2ordered(x: long): Ordered[long] = ...
implicit def float2ordered(x: float): Ordered[float] = ...
implicit def double2ordered(x: double): Ordered[double] = ...
implicit def boolean2ordered(x: boolean): Ordered[boolean] = ...
implicit def seq2ordered[A <% Ordered[A]](xs: Array[A]): Ordered[Seq[A]] =
  new Ordered[Seq[A]] with Proxy {
    def compare[B >: Seq[A] <% Ordered[B]](that: B): Int = that match {
      case that: Seq[A] =>
        var res = 0
        val these = this.elements
        val those = that.elements
        while (res == 0 && these.hasNext)
          res = if (!those.hasNext) 1 else these.next compare those.next
      case _ => - (that compare xs)
  }

implicit def string2ordered(x: String): Ordered[String] =
  new Ordered[String] with Proxy {
    def self: Any = x
    def compare [b >: String <% Ordered[b]](y: b): int = y match {
      case y1: String => x compare y1
      case _ => -(y compare x)
    }
  }

implicit def tuple2ordered[a1 <% Ordered[a1], a2 <% Ordered[a2]]
                          (x: Tuple2[a1, a2]): Ordered[Tuple2[a1, a2]] =
  new Ordered[Tuple2[a1, a2]] with Proxy {
    def self: Any = x
    def compare[T >: Tuple2[a1, a2] <% Ordered[T]](y: T): Int = y match {
      case y: Tuple2[a1, a2] =>
        val res = x._1 compare y._1
        if (res == 0) x._2 compare y._2
        else res
      case _ => -(y compare x)
    }
  }

// Analogous for Tuple3 to Tuple9

// Views into class Seq

implicit def string2seq(str: String): Seq[Char] = new Seq[Char] {
  def length = str.length()
  def elements = Iterator.fromString(str)
  def apply(n: Int) = str.charAt(n)
  override def hashCode(): Int = str.hashCode()
  override def equals(y: Any): Boolean = (str == y)
  override protected def stringPrefix: String = "String"
}
```

```
// Views from primitive types to Java's boxed types

implicit def byte2Byte(x: byte) = new java.lang.Byte(x)
implicit def short2Short(x: short) = new java.lang.Short(x)
implicit def char2Character(x: char) = new java.lang.Character(x)
implicit def int2Integer(x: int) = new java.lang.Integer(x)
implicit def long2Long(x: long) = new java.lang.Long(x)
implicit def float2Float(x: float) = new java.lang.Float(x)
implicit def double2Double(x: double) = new java.lang.Double(x)
implicit def boolean2Boolean(x: boolean) = new java.lang.Boolean(x)

// Numeric conversion views

implicit def byte2short(x: byte): short = x.toShort
implicit def byte2int(x: byte): int = x.toInt
implicit def byte2long(x: byte): long = x.toLong
implicit def byte2float(x: byte): float = x.toFloat
implicit def byte2double(x: byte): double = x.toDouble

implicit def short2int(x: short): int = x.toInt
implicit def short2long(x: short): long = x.toLong
implicit def short2float(x: short): float = x.toFloat
implicit def short2double(x: short): double = x.toDouble

implicit def char2int(x: char): int = x.toInt
implicit def char2long(x: char): long = x.toLong
implicit def char2float(x: char): float = x.toFloat
implicit def char2double(x: char): double = x.toDouble

implicit def int2long(x: int): long = x.toLong
implicit def int2float(x: int): float = x.toFloat
implicit def int2double(x: int): double = x.toDouble

implicit def long2float(x: long): float = x.toFloat
implicit def long2double(x: long): double = x.toDouble

implicit def float2double(x: float): double = x.toDouble
```

```
// Errors and asserts -----------------------------------------

  def error(message: String): Nothing = throw new Error(message)

  def exit(): Nothing = exit(0)
  def exit(status: Int): Nothing = {
    java.lang.System.exit(status)
    throw new Throwable()
  }

  def assert(assertion: Boolean): Unit =
    if (!assertion)
      throw new Error("assertion failed")

  def assert(assertion: Boolean, message: Any): Unit =
    if (!assertion)
      throw new Error("assertion failed: " + message)

  def assume(assumption: Boolean): Unit =
    if (!assumption)
      throw new Error("assumption failed")

  def assume(assumption: Boolean, message: Any): Unit =
    if (!assumption)
      throw new Error("assumption failed: " + message)
}
```

Bibliography

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs, 2nd edition.* MIT Press, Cambridge, Massachusetts, 1996.

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

[KP07] Andrew J. Kennedy and Benjamin C. Pierce. On Decidability of Nominal Subtyping with Variance, January 2007. FOOL-WOOD '07.

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

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[Oa04] Martin Odersky and al. An Overview of the Scala Programming Language. Technical Report IC/2004/64, EPFL Lausanne, Switzerland, 2004.

[OCRZ03a] 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`.

[OCRZ03b] Martin Odersky, Vincent Cremet, Christine Röckl, and Matthias Zenger. A Nominal Theory of Objects with Dependent Types. In *Proc. ECOOP'03*, Springer LNCS, July 2003.

[Ode06] Martin Odersky. The Scala Experiment – Can We Provide Better Language Support for Component Systems? In *Proc. ACM Symposium on Principles of Programming Languages*, 2006.

[OZ05a] Martin Odersky and Matthias Zenger. Independently Extensible Solutions to the Expression Problem. In *Proc. FOOL 12*, January 2005. `http://homepages.inf.ed.ac.uk/wadler/fool`.

[OZ05b] Martin Odersky and Matthias Zenger. Scalable Component Abstractions. In *Proc. OOPSLA*, 2005.

[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`.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Comm. ACM*, 20:822–823, November 1977.

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’
opchar           ::=  “all other characters in\u0020-007F and Unicode categories
                      Sm, So except parentheses ([]) and periods”

op               ::=  opchar {opchar}
varid            ::=  lower idrest
plainid          ::=  upper idrest
                   |  varid
                   |  op
id               ::=  plainid
                   |  ‘\‘’ stringLit ‘\‘’
idrest           ::=  {letter | digit} [‘_’ op]

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

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

```
booleanLiteral   ::=  true | false

characterLiteral ::= ‘\’’ printableChar ‘\’’
                   |  ‘\’’ charEscapeSeq ‘\’’

stringLiteral    ::=  ‘"’ {stringElement} ‘"’
                   |  ‘"""’ multiLineChars ‘"""’
stringElement    ::=  printableCharNoDoubleQuote
                   |  charEscapeSeq
multiLineChars   ::=  {[’"’] [’"’] charNoDoubleQuote}

symbolLiteral    ::=  ‘’’ plainid

comment          ::=  ‘/*’  “any sequence of characters”  ‘*/’
                   |  ‘//’  “any sequence of characters up to end of line”

nl               ::=  “new line character”
semi             ::=  ‘;’ |  nl {nl}
```

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

```
Literal           ::=  integerLiteral
                    |  floatingPointLiteral
                    |  booleanLiteral
                    |  characterLiteral
                    |  stringLiteral
                    |  symbolLiteral
                    |  null

QualId            ::=  id {‘.’ id}
ids               ::=  id {‘,’ id}

Path              ::=  StableId
                    |  [id ‘.’] this
StableId          ::=  id
                    |  Path ‘.’ id
                    |  [id ’.’] super [ClassQualifier] ‘.’ id
ClassQualifier    ::=  ‘[’ id ‘]’

Type              ::=  InfixType [‘=>’ Type]
                    |  ‘(’ [‘=>’ Type] ‘)’ ‘=>’ Type
InfixType         ::=  CompoundType {id [nl] CompoundType}
CompoundType      ::=  AnnotType {with AnnotType} [Refinement]
AnnotType         ::=  {Annotation} SimpleType
SimpleType        ::=  SimpleType TypeArgs
                    |  SimpleType ‘#’ id
                    |  StableId
                    |  Path ‘.’ type
                    |   ‘(’ Types [‘,’] ’)’
TypeArgs          ::=   ‘[’ Types ‘]’
Types             ::=   Type {‘,’ Type}
Refinement        ::=   [nl] ‘{’ RefineStat {semi RefineStat} ‘}’
RefineStat        ::=   Dcl
                    |   type TypeDef
                    |
TypePat           ::=   CompoundTypePat {id [nl] CompoundTypePat}
CompoundTypePat   ::=   AnnotTypePat {with AnnotTypePat}
AnnotTypePat      ::=   {Annotation} SimpleTypePat
SimpleTypePat     ::=   SimpleTypePat1 [TypePatArgs]
SimpleTypePat1    ::=   SimpleTypePat1 ‘#’ id
                    |   StableId
                    |   Path ‘.’ type
                    |   ‘(’ ArgTypePats [‘,’] ‘)’

TypePatArgs       ::=   ‘[’ ArgTypePats ’]’
ArgTypePats       ::=   ArgTypePat {‘,’ ArgTypePat}
ArgTypePat        ::=   varid
                    |   ‘_’
                    |   Type

Ascription        ::=   ‘:’ CompoundType
                    |   ‘:’ Annotation {Annotation}
                    |   ‘:’ ‘_’ ‘*’

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

```
                  |  Path
                  |  ‘_’
                  |  ‘(’ [Exprs [‘,’]] ‘)’
                  |  SimpleExpr ‘.’ id
                  |  SimpleExpr TypeArgs
                  |  SimpleExpr1 ArgumentExprs
                  |  XmlExpr
Exprs            ::=  Expr {‘,’ Expr}
ArgumentExprs    ::=  ‘(’ [Exprs [‘,’]] ’)’
                  |  [nl] BlockExpr
BlockExpr        ::=  ‘{’ CaseClauses ‘}’
                  |  ‘{’ Block ‘}’
Block            ::=  {BlockStat semi} [ResultExpr]
BlockStat        ::=  Import
                  |  [implicit] Def
                  |  {LocalModifier} TmplDef
                  |  Expr1
                  |
ResultExpr       ::=  Expr1
                  |  (Bindings | id ‘:’ CompoundType) ‘=>’ Block

Enumerators      ::=  Generator {semi Enumerator}
Enumerator       ::=  Generator
                  |  Guard
                  |  val Pattern1 ‘=’ Expr
Generator        ::=  Pattern1 ‘<-’ Expr [Guard]

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

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

```
TypeParamClause   ::=  '[' VariantTypeParam {',' VariantTypeParam} ']'
FunTypeParamClause::=  '[' TypeParam {',' TypeParam} ']'
VariantTypeParam  ::=  ['+' | '-'] TypeParam
TypeParam         ::=  id [>: Type] [<: Type] [<% Type]
ParamClauses      ::=  {ParamClause} [[nl] '(' implicit Params ')']
ParamClause       ::=  [nl] '(' [Params] ')'}
Params            ::=  Param {',' Param}
Param             ::=  {Annotation} id [':' ParamType]
ParamType         ::=  Type
                    |  '=>' Type
                    |  Type '*'
ClassParamClauses ::=  {ClassParamClause}
                       [[nl] '(' implicit ClassParams ')']
ClassParamClause  ::=  [nl] '(' [ClassParams] ')'
ClassParams       ::=  ClassParam {'' ClassParam}
ClassParam        ::=  {Annotation} [{Modifier} ('val' | 'var')]
                       id [':' ParamType]
Bindings          ::=  '(' Binding {',' Binding ')'
Binding           ::=  id [':' Type]

Modifier          ::=  LocalModifier
                    |  AcessModifier
                    |  override
LocalModifier     ::=  abstract
                    |  final
                    |  sealed
                    |  implicit
AccessModifier    ::=  (private | protected) [AccessQualifier]
AccessQualifier   ::=  '[' (id | this) ']'

Annotation        ::=  '@' AnnotationExpr [nl]
AnnotationExpr    ::=  Constr [[nl] '{' {NameValuePair} '}']
NameValuePair     ::=  val id '=' PrefixExpr

TemplateBody      ::=  [nl] '{' [id [':' Type] '=>']
                       TemplateStat {semi TemplateStat} '}'
TemplateStat      ::=  Import
                    |  {Annotation} {Modifier} Def
                    |  {Annotation} {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 {nl} TypeDcl

ValDcl           ::=  ids ‘:’ Type
VarDcl           ::=  ids ‘:’ Type
FunDcl           ::=  FunSig [‘:’ Type]
FunSig           ::=  id [FunTypeParamClause] ParamClauses
TypeDcl          ::=  id [‘>:’ Type] [‘<:’ Type]

Def              ::=  val PatDef
                  |  var VarDef
                  |  def FunDef
                  |  type {nl} TypeDef
                  |  TmplDef
PatDef           ::=  Pattern2 {‘,’ Pattern2} [‘:’ Type] ‘=’ Expr
VarDef           ::=  ids [‘:’ Type] ‘=’ Expr
                  |  ids ‘:’ Type ‘=’ ‘_’
FunDef           ::=  FunSig ‘:’ Type ‘=’ Expr
                  |  FunSig [nl] ‘{’ Block ‘}’
                  |  this ParamClause ParamClauses
                      (‘=’ ConstrExpr | [nl] ConstrBlock)
TypeDef          ::=  id [TypeParamClause] ‘=’ Type

TmplDef          ::=  [case] class ClassDef
                  |  [case] object ObjectDef
                  |  trait TraitDef
ClassDef         ::=  id [TypeParamClause] {Annotation} [AccessModifier]
                      ClassParamClauses [requires AnnotType] ClassTemplateOpt
TraitDef         ::=  id [TypeParamClause] [requires AnnotType] TraitTemplateOpt
ObjectDef        ::=  id ClassTemplateOpt
ClassTemplateOpt ::=  extends ClassTemplate | [[extends] TemplateBody]
TraitTemplateOpt ::=  extends TraitTemplate | [[extends] TemplateBody]
ClassTemplate    ::=  [EarlyDefs] ClassParents [TemplateBody]
TraitTemplate    ::=  [EarlyDefs] TraitParents [TemplateBody]
ClassParents     ::=  Constr {with AnnotType}
TraitParents     ::=  AnnotType {with AnnotType}
Constr           ::=  AnnotType {ArgumentExprs}
EarlyDefs        ::= ‘{’ [EarlyDef {semi EarlyDef}] ‘}’ with
EarlyDef         ::=  Annotations Modifiers PatDef

ConstrExpr       ::=  SelfInvocation
                  |  ConstrBlock
ConstrBlock      ::=  ‘{’ SelfInvocation {semi BlockStat} ‘}’
SelfInvocation   ::=  this ArgumentExprs {ArgumentExprs}

TopStatSeq       ::=  TopStat {semi TopStat}
TopStat          ::=  {Annotation} {Modifier} TmplDef
                  |  Import
```

```
                        |  Packaging
                        |
Packaging         ::=  **package** QualId [nl] ‘{’ TopStatSeq ‘}’

CompilationUnit   ::=  [**package** QualId semi] TopStatSeq
```

Chapter B

Change Log

Changes in Version 2.5.0

Type constructor polymorphism[1]

Type parameters (§20.4) and abstract type members (§20.3) can now also abstract over type constructors (§19.3.3).

This allows a more precise *Iterable* interface:

```
trait Iterable[+t] {
  type MyType[+t] <: Iterable[t] // MyType is a type constructor

  def filter(p: t => Boolean): MyType[t] = ...
  def map[s](f: t => s): MyType[s] = ...
}

abstract class List[+t] extends Iterable[t] {
  type MyType[+t] = List[t]
}
```

This definition of *Iterable* makes explicit that mapping a function over a certain structure (e.g., a *List*) will yield the same structure (containing different elements).

Early object initialization

It is now possible to initialize some fields of an object before any parent constructors are called (§21.1.6). This is particularly useful for traits, which do not have normal constructor parameters. Example:

```
trait Greeting {
```

[1]Implemented by Adriaan Moors

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

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

For-comprehensions, revised

The syntax of for-comprehensions has changed (§22.18). In the new syntax, generators do not start with a **val** anymore, but filters start with an **if** (and are called guards). A semicolon in front of a guard is optional. For example:

```
  for (val x <- List(1, 2, 3); x % 2 == 0) println(x)
```

is now written

```
  for (x <- List(1, 2, 3) if x % 2 == 0) println(x)
```

The old syntax is still available but will be deprecated in the future.

Implicit anonymous functions

It is now possible to define anonymous functions using underscores in parameter position (§Example 22.22.1). For instance, the expressions in the left column are each function values which expand to the anonymous functions on their right.

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

As a special case (§22.6), a partially unapplied method is now designated `m _` instead of the previous notation `&m`.

The new notation will displace the special syntax forms `.m()` for abstracting over method receivers and `&m` for treating an unapplied method as a function value. For the time being, the old syntax forms are still available, but they will be deprecated in the future.

Pattern matching anonymous functions, refined

It is now possible to use case clauses to define a function value directly for functions of arities greater than one (§24.5). Previously, only unary functions could be defined that way. Example:

```
def scalarProduct(xs: Array[Double], ys: Array[Double]) =
  (0.0 /: (xs zip ys)) {
    case (a, (b, c)) => a + b * c
  }
```

Changes in Version 2.4.0

Object-local private and protected

The **private** and **protected** modifiers now accept a `[`**this**`]` qualifier (§21.2). A definition M which is labelled **private**`[`**this**`]` is private, and in addition can be accessed only from within the current object. That is, the only legal prefixes for M are **this** or C**.this**. Analogously, a definition M which is labelled **protected**`[`**this**`]` is protected, and in addition can be accessed only from within the current object.

Tuples, revised

The syntax for tuples has been changed from {…} to (…) (§22.8). For any sequence of types $T_1, \ldots, T_n$,

$(T_1, \ldots, T_n)$ is a shorthand for `Tuple`n`[`$T_1, \ldots, T_n$`]`.

Analogously, for any sequence of expressions or patterns $x_1, \ldots, x_n$,

$(x_1, \ldots, x_n)$ is a shorthand for `Tuple`n`(`$x_1, \ldots, x_n$`)`.

Access modifiers for primary constructors

The primary constructor of a class can now be marked **private** or **protected** (§21.3). If such an access modifier is given, it comes between the name of the class and its value parameters. Example:

```
class C[T] private (x: T) { ... }
```

Annotations

The support for attributes has been extended and its syntax changed (§27). Attributes are now called *annotations*. The syntax has been changed to follow Java's conventions, e.g. `@attribute` instead of `[attribute]`. The old syntax is still available but will be deprecated in the future.

Annotations are now serialized so that they can be read by compile-time or run-time tools. Class `scala.Annotation` has two sub-traits which are used to indicate how annotations are retained. Instances of an annotation class inheriting from trait `scala.ClassfileAnnotation` will be stored in the generated class files. Instances of an annotation class inheriting from trait `scala.StaticAnnotation` will be visible to the Scala type-checker in every compilation unit where the annotated symbol is accessed.

Decidable subtyping

The implementation of subtyping has been changed to prevent infinite recursions. Termination of subtyping is now ensured by a new restriction of class graphs to be finitary (§21.1.5).

Case classes cannot be abstract

It is now explicitly ruled out that case classes can be abstract (§21.2). The specification was silent on this point before, but did not explain how abstract case classes were treated. The Scala compiler allowed the idiom.

New syntax for self aliases and self types

It is now possible to give an explicit alias name and/or type for the self reference **`this`** (§21.1). For instance, in

```
class C { self: D =>
  ...
}
```

the name `self` is introduced as an alias for **`this`** within `C` and the self type (§21.3) of `C` is assumed to be `D`. This construct is introduced now in order to replace eventually both the qualified this construct `C.`**`this`** and the **`requires`** clause in Scala.

Assignment Operators

It is now possible to combine operators with assignments (§22.11.4). Example:

```
var x: int = 0
x += 1
```

Changes in Version 2.3.2 (23-Jan-2007)

Extractors

It is now possible to define patterns independently of case classes, using `unapply` methods in extractor objects (§24.1.7). Here is an example:

```
object Twice {
  def apply(x:Int): int = x*2
  def unapply(z:Int): Option[int] = if (z%2==0) Some(z/2) else None
}
val x = Twice(21)
x match { case Twice(n) => Console.println(n) } // prints 21
```

In the example, `Twice` is an extractor object with two methods:

- The `apply` method is used to build even numbers.
- The `unapply` method is used to decompose an even number; it is in a sense the reverse of `apply`. `unapply` methods return option types: `Some(...)` for a match that suceeds, `None` for a match that fails. Pattern variables are returned as the elements of `Some`. If there are several variables, they are grouped in a tuple.

In the second-to-last line, `Twice`'s `apply` method is used to construct a number `x`. In the last line, `x` is tested against the pattern `Twice(n)`. This pattern succeeds for even numbers and assigns to the variable `n` one half of the number that was tested. The pattern match makes use of the `unapply` method of object `Twice`. More details on extractors can be found in the paper "Matching Objects with Patterns" by Emir, Odersky and Williams.

Tuples

A new lightweight syntax for tuples has been introduced (§22.8). For any sequence of types $T_1, \ldots, T_n$,

$\{T_1, \ldots, T_n\}$ is a shorthand for $\texttt{Tuple}n\texttt{[}T_1, \ldots, T_n\texttt{]}$.

Analogously, for any sequence of expressions or patterns $x_1, \ldots, x_n$,

$\{x_1, \ldots, x_n\}$ is a shorthand for $\texttt{Tuple}n\texttt{(}x_1, \ldots, x_n\texttt{)}$.

Infix operators of greater arities

It is now possible to use methods which have more than one parameter as infix operators (§22.11). In this case, all method arguments are written as a normal parameter list in parentheses. Example:

```
class C {
  def +(x: int, y: String) = ...
}
val c = new C
c + (1, "abc")
```

Deprecated attribute

A new standard attribute `deprecated` is available (§27). If a member definition is marked with this attribute, any reference to the member will cause a "deprecated" warning message to be emitted.

Changes in Version 2.3.0 (23-Nov-2006)

Procedures

A simplified syntax for functions returning `unit` has been introduced (§20.6.3). Scala now allows the following shorthands:

```
def f(params)                 for     def f(params): unit
def f(params) { ... }         for     def f(params): unit = { ... }
```

Type Patterns

The syntax of types in patterns has been refined (§24.2). Scala now distinguishes between type variables (starting with a lower case letter) and types as type arguments in patterns. Type variables are bound in the pattern. Other type arguments are, as in previous versions, erased. The Scala compiler will now issue an "unchecked" warning at places where type erasure might compromise type-safety.

Standard Types

The recommended names for the two bottom classes in Scala's type hierarchy have changed as follows:

```
All      ==>     Nothing
AllRef   ==>     Null
```

The old names are still available as type aliases.

Changes in Version 2.1.8 (23-Aug-2006)

Visibility Qualifier for protected

Protected members can now have a visibility qualifier (§21.2), e.g. **protected**[<qualifier>]. In particular, one can now simulate package protected access as in Java writing

```
protected[P] def X ...
```

where P would name the package containing X.

Relaxation of Private Acess

Private members of a class can now be referenced from the companion module of the class and vice versa (§21.2)

Implicit Lookup

The lookup method for implicit definitions has been generalized (§23.2). When searching for an implicit definition matching a type T, now are considered

1. all identifiers accessible without prefix, and
2. all members of companion modules of classes associated with T.

(The second clause is more general than before). Here, a class is *associated* with a type T if it is referenced by some part of T, or if it is a base class of some part of T. For instance, to find implicit members corresponding to the type

```
HashSet[List[Int], String]
```

one would now look in the companion modules (aka static parts) of `HashSet`, `List`, `Int`, and `String`. Before, it was just the static part of `HashSet`.

Tightened Pattern Match

A typed pattern match with a singleton type `p.type` now tests whether the selector value is reference-equal to p (§24.1). Example:

```
val p = List(1, 2, 3)
val q = List(1, 2)
val r = q
r match {
  case _: p.type => Console.println("p")
  case _: q.type => Console.println("q")
}
```

This will match the second case and hence will print `"q"`. Before, the singleton types were erased to `List`, and therefore the first case would have matched, which is non-sensical.

Changes in Version 2.1.7 (19-Jul-2006)

Multi-Line string literals

It is now possible to write multi-line string-literals enclosed in triple quotes (§17.3.5). Example:

```
"""this is a
   multi-line
   string literal"""
```

No escape substitutions except for unicode escapes are performed in such string literals.

Closure Syntax

The syntax of closures has been slightly restricted (§22.22). The form

```
x: T => E
```

is valid only when enclosed in braces, i.e. `{ x: T => E }`. The following is illegal, because it might be read as the value x typed with the type T => E:

```
val f = x: T => E
```

Legal alternatives are:

```
val f = { x: T => E }
val f = (x: T) => E
```

Changes in Version 2.1.5 (24-May-2006)

Class Literals

There is a new syntax for class literals (§22.1): For any class type C, `classOf[`C`]` designates the run-time representation of C.

Changes in Version 2.0 (12-Mar-2006)

Scala in its second version is different in some details from the first version of the language. There have been several additions and some old idioms are no longer supported. This appendix summarizes the main changes.

New Keywords

The following three words are now reserved; they cannot be used as identifiers (§17.1)

```
implicit   match    requires
```

Newlines as Statement Separators

Newlines can now be used as statement separators in place of semicolons (§17.2)

Syntax Restrictions

There are some other situations where old constructs no longer work:

Pattern matching expressions. The **match** keyword now appears only as infix operator between a selector expression and a number of cases, as in:

```
expr match {
  case Some(x) => ...
  case None => ...
}
```

Variants such as `expr.match {...}` or just `match {...}` are no longer supported.

"With" in extends clauses. . The idiom

```
class C with M { ... }
```

is no longer supported. A **with** connective is only allowed following an **extends** clause. For instance, the line above would have to be written

```
class C extends AnyRef with M { ... } .
```

However, assuming `M` is a trait (see 21.3.3), it is also legal to write

```
class C extends M { ... }
```

The latter expression is treated as equivalent to

```
class C extends S with M { ... }
```

where S is the superclass of M.

Regular Expression Patterns. The only form of regular expression pattern that is currently supported is a sequence pattern, which might end in a sequence wildcard _*. Example:

```
case List(1, 2, _*) => ... // will match all lists starting with \code{1,2}.
```

It is at current not clear whether this is a permanent restriction. We are evaluating the possibility of re-introducing full regular expression patterns in Scala.

Selftype Annotations

The recommended syntax of selftype annotations has changed.

```
class C: T extends B { ... }
```

becomes

```
class C requires T extends B { ... }
```

That is, selftypes are now indicated by the new **requires** keyword. The old syntax is still available but is considered deprecated. Conversions

For-comprehensions

For-comprehensions (§22.18) now admit value and pattern definitions. Example:

```
for {
  val x <- List.range(1, 100)
  val y <- List.range(1, x)
  val z = x + y
  isPrime(z)
} yield Pair(x, y)
```

Note the definition `val z = x + y` as the third item in the for-comprehension.

Conversions

The rules for implicit conversions of methods to functions (§22.24) have been tightened. Previously, a parameterized method used as a value was always implicitly converted to a function. This could lead to unexpected results when method arguments where forgotten. Consider for instance the statement below:

```
show(x.toString)
```

where show is defined as follows:

```
def show(x: String) = Console.println(x) .
```

Most likely, the programmer forgot to supply an empty argument list `()` to `toString`. The previous Scala version would treat this code as a partially applied method, and expand it to:

```
show(() => x.toString())
```

As a result, the address of a closure would be printed instead of the value of s.

Scala version 2.0 will apply a conversion from partially applied method to function value only if the expected type of the expression is indeed a function type. For instance, the conversion would not be applied in the code above because the expected type of show's parameter is `String`, not a function type.

The new convention disallows some previously legal code. Example:

```
def sum(f: int => double)(a: int, b: int): double =
  if (a > b) 0 else f(a) + sum(f)(a + 1, b)

val sumInts  =  sum(x => x)  // error: missing arguments
```

The partial application of sum in the last line of the code above will not be converted to a function type. Instead, the compiler will produce an error message which states that arguments for method sum are missing. The problem can be fixed by providing an expected type for the partial application, for instance by annotating the definition of `sumInts` with its type:

```
val sumInts: (int, int) => double  =  sum(x => x)  // OK
```

On the other hand, Scala version 2.0 now automatically applies methods with empty parameter lists to `()` argument lists when necessary. For instance, the show expression above will now be expanded to

```
show(x.toString()) .
```

Scala version 2.0 also relaxes the rules of overriding with respect to empty parameter lists. The revised definition of *matching members* (§21.1.3) makes it now possible to override a method with an explicit, but empty parameter list `()` with a parameterless method, and *vice versa*. For instance, the following class definition is now legal:

```
class C {
  override def toString: String = ...
}
```

Previously this definition would have been rejected, because the `toString` method as inherited from `java.lang.Object` takes an empty parameter list.

Class Parameters

A class parameter may now be prefixed by **val** or **var** (§21.3).

Private Qualifiers

Previously, Scala had three levels of visibility: *private*, *protected* and *public*. There was no way to restrict accesses to members of the current package, as in Java. Scala 2 now defines access qualifiers that let one express this level of visibility, among others. In the definition

```
private[C] def f(...)
```

access to `f` is restricted to all code within the class or package `C` (which must contain the definition of `f`) (§21.2)

Changes in the Mixin Model

The model which details mixin composition of classes has changed significantly. The main differences are:

1. We now distinguish between *traits* that are used as mixin classes and normal classes. The syntax of traits has been generalized from version 1.0, in that traits are now allowed to have mutable fields. However, as in version 1.0, traits may still do not have constructor parameters.
2. Member resolution and super accesses are now both defined in terms of a *class linearization*.
3. Scala's notion of method overloading has been generalized; in particular, it is now possible to have overloaded variants of the same method in a subclass and in a superclass, or in several different mixins. This makes method overloading in Scala conceptually the same as in Java.

The new mixin model is explained in more detail in §21.

Implicit Parameters

Views in Scala 1.0 have been replaced by the more general concept of implicit parameters (§23)

Flexible Typing of Pattern Matching

The new version of Scala implements more flexible typing rules when it comes to pattern matching over heterogeneous class hierarchies (§24.4). A *heterogeneous class hierarchy* is one where subclasses inherit a common superclass with different parameter types. With the new rules in Scala version 2.0 one can perform pattern matches over such hierarchies with more precise typings that keep track of the information gained by comparing the types of a selector and a matching pattern (§Example 24.4.1). This gives Scala capabilities analogous to guarded algebraic data types.