Scala By Example

DRAFT
August 23, 2007

Martin Odersky

PROGRAMMING METHODS LABORATORY
EPFL
SWITZERLAND

Contents

Chapter 1

Introduction

Scala smoothly integrates object-oriented and functional programming. It is designed to express common programming patterns in a consise, elegant, and type-safe way. Scala introduces several innovative language constructs. For instance:

- Abstract types and mixin composition unify concepts from object and module systems.
- Pattern matching over class hierarchies unifies functional and object-oriented data access. It greatly simplifies the processing of XML trees.
- A flexible syntax and type system enables the construction of advanced libraries and new domain specific languages.

At the same time, Scala is compatible with Java. Java libraries and frameworks can be used without glue code or additional declarations.

This document introduces Scala in an informal way, through a sequence of examples.

Chapters 2 and 3 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 meant to be complemented by the Scala Language Reference Manual which specifies Scala in a more detailed and precise way.

Acknowledgment. We owe a great debt to Abelson's and Sussman's wonderful book "Structure and Interpretation of Computer Programs"[ASS96]. Many of their examples and exercises are also present here. 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.

Chapter 2

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 += 1
      while (xs(j) > pivot) j -= 1
      if (i <= j) {
        swap(i, j)
        i += 1
        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 3

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 3.1.

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 3.2 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 3.1: 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 3.2: 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 18.11.

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 4

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.

4.1 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. For example:

```
scala> 87 + 145
unnamed0: Int = 232

scala> 5 + 2 * 3
unnamed1: Int = 11

scala> "hello" + " world!"
unnamed2: java.lang.String = hello world!
```

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

```
scala> def scale = 5
scale: Int

scala> 7 * scale
unnamed3: Int = 35

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

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

scala> 2 * pi * radius
unnamed4: Double = 62.83185307179586
```

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 4.1.1 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*.

4.2 Parameters

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

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

scala> square(2)
unnamed0: Double = 4.0

scala> square(5 + 3)
unnamed1: Double = 64.0

scala> square(square(4))
unnamed2: Double = 256.0

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

scala> sumOfSquares(3, 2 + 2)
unnamed3: Double = 25.0
```

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 4.2.1

```
    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:

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

scala> def first(x: Int, y: Int) = x
first: (Int,Int)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 4.2.2

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

scala> constOne(1, loop)
unnamed0: Int = 1

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

```
^C                                        // stops execution with Ctrl-C
```

4.3 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 4.3.1

```
scala> def abs(x: Double) = if (x >= 0) x else -x
abs: (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 `||`.

4.4 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 $\quad$ x/y $\quad$ $(y + x/y)/2$

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 4.4.1 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 4.4.2 Trace the execution of the `sqrt(4)` expression.

4.5 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)
}
```

4.6 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 4.6.1 Design a tail-recursive version of `factorial`.

Chapter 5

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 2 * 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): Int =
  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 2 * powerOfTwo(x - 1)
```

5.1 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 '=>' are the parameters of the function, whereas the part following the '=>' is its body. 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".

5.2 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,

```
scala> sumSquares(1, 10) + sumPowersOfTwo(10, 20)
unnamed0: Int = 267632001
```

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 (args_1) ... (args_n) = E
```

is equivalent to

```
def f = (args_1) => ... => (args_n) => E .
```

Or, equivalently, using a value definition:

```
val f = (args_1) => ... => (args_n) => 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.

```
T_1 => T_2 => T_3       is equivalent to     T_1 => (T_2 => T_3)
```

Exercise 5.2.1 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 5.2.2 Write a function `product` that computes the product of the values of functions at points over a given range.

Exercise 5.2.3 Write `factorial` in terms of `product`.

Exercise 5.2.4 Can you write an even more general function which generalizes both `sum` and `product`?

5.3 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)
    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:

```
scala> def sqrt(x: Double) = fixedPoint(y => (y + x/y) / 2)(1.0)
sqrt: (Double)Double

scala> 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 4.4.

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

5.4 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.

5.5 Language Elements Seen So Far

Chapters 4 and 5 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`, `1.0e10`, `'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 | Float | Double |
                 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 6

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 += new Rational(1, i)
  i += 1
}
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)
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 6.0.1 Write methods `union` and `intersection` to form the union and intersection between two sets.

Exercise 6.0.2 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 8.6).

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`. For example:

```
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 toLong: Long
  def toFloat: Float
  def toDouble: 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`:

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

The implementation of the `Zero` object is straightforward:

```
object Zero extends Nat {
  def isZero: Boolean = true
  def predecessor: Nat = error("negative number")
  def successor: Nat = new Succ(Zero)
  def + (that: Nat): Nat = that
  def - (that: Nat): Nat = if (that.isZero) Zero
                           else 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 6.0.3 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 7

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 = error("Number.leftOp")
  def rightOp: Expr = error("Number.rightOp")
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def isNumber: Boolean = false
  def isSum: Boolean = true
```

```
  def numValue: Int = 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 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) Console.print(e.numValue)
  else if (e.isSum) {
    Console.print("(")
    print(e.leftOp)
    Console.print("+")
    print(e.rightOp)
    Console.print(")")
  } else 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 { Console.print(n) }
}
class Sum(e1: Expr, e2: Expr) extends Expr {
  def eval: Int = e1.eval + e2.eval
  def print {
    Console.print("(")
    print(e1)
    Console.print("+")
    print(e2)
    Console.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.

7.1 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.

7.2 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 7.2.1 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 7.2.2 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 P1 => E1 ... case Pn => En }
```

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 P1 => E1 ... case Pn => En })
```

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

Chapter 8

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 = error("EmptyStack.top")
  def pop = 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 = error("EmptyStack.top")
  def pop = 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)
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(s1.top)
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.

8.1 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 ex-

pected 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[Num].incl(Num(1.0)).incl(Num(2.0))
s.contains(Num(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 15.

8.2 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 `String`s is a special case of a stack of `AnyRef`s. 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.

8.3 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 >: S`, the type parameter `T` is restricted to range only over *supertypes* of type `S`. (One can also combine lower and upper bounds, as in `T >: S <: 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.

8.4 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 = error("EmptyStack.top")
  def pop = 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.

8.5 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)
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) =>
    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) => println("quotient: " + n + ", rest: " + d)
}
```

8.6 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 8.2 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 8.6.1 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 9

Lists

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

9.1 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 9.1.1 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)
}
```

9.2 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 => error("Nil.head")
  case x :: xs => x
}
def tail: List[A] = this match {
```

```
    case Nil => error("Nil.tail")
    case x :: xs => xs
  }
```

The next function computes the length of a list.

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

Exercise 9.2.1 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      => 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

$$n + (n-1) + \ldots + 1 = n(n+1)/2$$

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.

9.3 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)
```

9.4 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(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 => println(x))
```

Exercise 9.4.1 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:

```
sum(List(x1, ..., xn))       =  0 + x1 + ... + xn
product(List(x1, ..., xn))   =  1 * x1 * ... * xn
```

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     => 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 => 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 9.4.2 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 9.2 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 9.4.3 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 $1 \leq j < i < n$. 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))
```

9.5 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 10

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)
```

10.1 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 10.1.1 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.

10.2 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)
```

10.3 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 10.3.1 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 10.3.2 Translate

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

to higher-order functions.

10.4 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) print(x + "\t")
  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`.

10.5 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 16.2 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 11

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.

11.1 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 += amount
  }

  def withdraw(amount: Int): Int =
    if (0 < amount && amount <= balance) {
      balance -= amount
      balance
    } else 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 11.1.1 Here is a `scalaint` session that deals with bank accounts.

```
scala> :l bankaccount.scala
Loading bankaccount.scala...
defined class BankAccount
scala> val account = new BankAccount
account: BankAccount = BankAccount$class@1797795
scala> account deposit 50
unnamed0: Unit = ()
scala> account withdraw 20
unnamed1: Int = 30
scala> account withdraw 20
unnamed2: Int = 10
scala> account withdraw 15
java.lang.Error: insufficient funds
        at scala.Predef$error(Predef.scala:74)
        at BankAccount$class.withdraw(<console>:14)
        at <init>(<console>:5)
scala>
```

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.

11.2 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 *= x
    i = i << 1
    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) {
  if (condition) {
    command; whileLoop(condition)(command)
  } else {}
}
```

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

Exercise 11.2.1 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 10.4.

11.3 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 11.3.1 Write the implementation of `orGate`.

Exercise 11.3.2 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.

```
  abstract class Simulation {
    case class WorkItem(time: Int, action: Action)
    private type Agenda = List[WorkItem]
    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, block)` inserts the element `WorkItem(currentTime + delay, () => block)` into the `agenda` list at the appropriate place.

```
private def insert(ag: Agenda, item: WorkItem): Agenda =
  if (ag.isEmpty || item.time < ag.head.time) item :: ag
  else ag.head :: insert(ag.tail, item)

def afterDelay(delay: Int)(block: => Unit) {
  val item = WorkItem(currentTime + delay, () => block)
  agenda = insert(agenda, item)
}
```

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

```
private def next() {
  agenda match {
    case WorkItem(time, action) :: rest =>
      agenda = rest; curtime = time; action()
    case List() =>
  }
}

def run() {
  afterDelay(0) { println("*** simulation started ***") }
  while (!agenda.isEmpty) next()
}
```

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 {
    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:

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

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

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

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

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

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

```
scala> input1 setSignal true; run
*** simulation started ***
sum 8 new_value = true

scala> input2 setSignal true; run
carry 11 new_value = true
sum 15 new_value = false
```

11.4 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 12

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 += 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) { Console.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 13

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.

```
val it: Iterator[Int] = Iterator.range(1, 100)
while (it.hasNext) {
  val x = it.next
  println(x * x)
}
```

13.1 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 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))
  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}
  }
}
```

13.2 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 = 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 += 1; x }
      else 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 += 1
      else 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 += 1; last }
}
```

13.3 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 => println(x))
```

Or, using a for-comprehension:

```
for (x <- Iterator.fromArray(xs))
  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 14

Lazy Values

Lazy values provide a way to delay initialization of a value until the first time it is accessed. This may be useful when dealing with values that might not be needed during execution, and whose computational cost is signifficant. As a first example, let's consider a database of employees, containing for each employee its manager and its team.

```
case class Employee(id: Int,
                    name: String,
                    managerId: Int) {
  val manager: Employee = Db.get(managerId)
  val team: List[Employee] = Db.team(id)
}
```

The `Employee` class given above will eagerly initialize all its fields, loading the whole employee table in memory. This is certainly not optimal, and it can be easily improved my making the fields lazy. This way we delay the database access until it is really needed, if it is ever needed.

```
case class Employee(id: Int,
                    name: String,
                    managerId: Int) {
  lazy val manager: Employee = Db.get(managerId)
  lazy val team: List[Employee] = Db.team(id)
}
```

To see what is really happening, we can use this mockup database which shows when records are fetched:

```
object Db {
  val table = Map(1 -> (1, "Haruki Murakami", -1),
                  2 -> (2, "Milan Kundera", 1),
```

```
                3 -> (3, "Jeffrey Eugenides", 1),
                4 -> (4, "Mario Vargas Llosa", 1),
                5 -> (5, "Julian Barnes", 2))

  def team(id: Int) = {
    for (rec <- table.values.toList; if rec._3 == id)
      yield recToEmployee(rec)
  }

  def get(id: Int) = recToEmployee(table(id))

  private def recToEmployee(rec: (Int, String, Int)) = {
    println("[db] fetching " + rec._1)
    Employee(rec._1, rec._2, rec._3)
  }
}
```

The output when running a program that retrieves one employee confirms that the database is only accessed when referring the lazy values.

Another use of lazy values is to resolve the initialization order of applications composed of several modules. Before lazy values were introduced, the same effect was achieved by using **object** definitions. As a second example, we consider a compiler composed of several modules. We look first at a simple symbol table that defines a class for symbols and two predefined functions.

```
class Symbols(val compiler: Compiler) {
  import compiler.types._

  val Add = new Symbol("+", FunType(List(IntType, IntType), IntType))
  val Sub = new Symbol("-", FunType(List(IntType, IntType), IntType))

  class Symbol(name: String, tpe: Type) {
    override def toString = name + ": " + tpe
  }
}
```

The `symbols` module is parameterized with a `Compiler` instance, which provides access to other services, such the the types module. In our example there are only two predefined functions, addition and subtraction, and their definitions depend on the `types` module.

```
class Types(val compiler: Compiler) {
  import compiler.symtab._

  abstract class Type
  case class FunType(args: List[Type], res: Type) extends Type
```

```
  case class NamedType(sym: Symbol) extends Type
  case object IntType extends Type
}
```

In order to hook the two components together a compiler object is created and passed as argument to the two components.

```
class Compiler {
  val symtab = new Symbols(this)
  val types  = new Types(this)
}
```

Unfortunately, the straight-forward approach fails at runtime because the `symtab` module needs the `types` module. In general, the dependency between modules can be complicated and getting the right initialization order is difficult, or even impossible when there are cycles. The easy fix is to make such fields `lazy` and let the compiler figure out the right order.

```
class Compiler {
  lazy val symtab = new Symbols(this)
  lazy val types  = new Types(this)
}
```

Now the two modules are initialized on first access, and the compiler may run as expected.

Syntax

The `lazy` modifier is allowed only on concrete value definitions. All typing rules for value definitions apply for `lazy` values as well, with one restriction removed: recursive local values are allowed.

Chapter 15

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 16

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

16.1 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 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 con-

stant 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) =>
        println("parsed: " + args(0).substring(0, n))
      case None =>
        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
```

16.2 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 10. 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 10. 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, _) => println("parsed: " + list)
      case None => println("nothing parsed")
    }
  }
}
```

Exercise 16.2.1 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 17

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 += 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, _) =>
          println("" + term + ": " + showType(term))
        case None =>
          println("syntax error")
      }
    }
  }// testInfer
```

To do the parsing, method main uses the combinator parser scheme of Chapter 16. 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 gram-

mar.

```
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 16 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 17.0.2 Using the parser library constructed in Exercise Exercise 16.2.1, modify the MiniML parser library so that no marker ";" is necessary for indicating the end of input.

Exercise 17.0.3 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 18

Abstractions for Concurrency

This section reviews common concurrent programming patterns and shows how they can be implemented in Scala.

18.1 Signals and Monitors

Example 18.1.1 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()
```

```
}
```

18.2 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
  }
}
```

18.3 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.

18.4 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
}
```

18.5 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()
  }
}
```

18.6 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 18.10).

```
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)
  }
}
```

18.7 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
  }
}
```

18.8 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
  }
}
```

18.9 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 18.3 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)
  println(f())
}
```

18.10 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.

18.11 Actors

Chapter 3 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)
}
```

Bibliography

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *The Structure and Interpretation of Computer Programs, 2nd edition*. MIT Press, Cambridge, Massachusetts, 1996.

[Mil78] Robin Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, Dec 1978.

[Wir77] Niklaus Wirth. What can we do about the unnecessary diversity of notation for syntactic definitions? *Comm. ACM*, 20:822–823, November 1977.