CS-210 Functional Programming Principles in Scala

SEPTEMBER 21, 2016

Table of Contents

1. Books

- Structure and Interpretation of Computer Programs, Harold Abelson and Gerald Jay Sussman, MIT Press
- Programming in Scala, Martin Odersky, Lex Spoon and Bill Venners, 2nd edition, Artima 2010

2. Call-by-name (CBN), call-by-value (CBV)

Let's say we have the following function, and that we call it in the following way:

```scala
def test(x: Int, y: int) = x * x

test(3+4, 2)
```

There are 2 strategies to solving this: send the function the uncalculated arguments (CBN) or calculate the arguments and *then* send them to the function (CBV).

- CBN and CBV reduce an expression to the same value as long as both evaluations terminate.
- If CBV evaluation of an expression e terminates, then CBN evaluation of e terminates too
- The other direction is not true.

Here's an example:

```scala
def first(x: Int, y: Int) = x
def loop: Int = loop

first(1, loop) // reduces to 1 under CBN since loop isn't run
first(1, loop) // does not terminate under CBV
```

Scala normally uses CBV, but you can force CBN with the `=>`.

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

def or(x: Boolean, y: => Boolean) = if (x) y else false // we need to return y as a va
```

2.1. Value definitions

Using `def` is CBN, but `val` is CBV.

```
val x = 2 // x refers to 2
val y = square(x) // y refers to 4, and not the function square(x)

def x = loop // OK
val x = loop // does not terminate since loop is evaluated
```

3. Blocks and lexical scope

To avoid namespace pollution, we can use nested functions:

```
def sqrt(x: Double) = {
    def sqrtIter(guess: Double): Double =
        if (isGoodEnough(guess)) guess
        else sqrtIter(improve(guess))

    def isGoodEnough(guess: Double) =
        abs(guess * guess - x) / x < 0.001

    def improve(guess: Double) =
        (guess + x / guess) / 2

    sqrtIter(1.0)
}
```

This is done using a block, delimited by `{ ... }` braces. The last element of a block is an expression that defines its return value.

The definitions inside a block are only visible from within the block. The block has access to what's been defined outside of it, but if it redefines an external definition, the new one will *shadow* the old one, meaning it will be redefined inside the block.

4. Tail recursion

If a function calls itself as its last action, then the function's stack frame can be reused. This is called *tail recursion*. In practice, this means that recursion is iterative in Scala, and is just as efficient as a loop.

One can require that a function is tail-recursive using a `@tailrec` annotation:

```
@tailrec
def gcd(a: Int, b:Int): Int = ...
```

An error is issued if `gcd` isn't tail recursive.

5. Higher-Order Functions

Functions that take other functions as parameters or that return functions as results are called *higher order functions*, as opposed to a *first order function* that acts on simple data types.

```
// Higher order function
// Corresponds to the sum of f(n) from a to b
def sum(f: Int => Int, a: Int, b: Int): =
    if (a > b) 0
    else f(a) + sum(f, a + 1, b)

// Different functions f
def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x

// Calling our higher order function
def sumInts(a: Int, b: Int): Int = sum(id, a, b)
def sumCubes(a: Int, b: Int): Int = sum(cube, a, b)
```

5.1. Anonymous functions

Instead of having to define a `cube` and `id` function in the example above, we can just write an anonymous function as such:

```
def sumInts(a: Int, b:Int): Int = sum(x => x, a, b)
def sumCubes(a: Int, b: Int): Int = sum(x => x*x*x, a, b)
```

6. Currying

From Wikipedia:

> *Currying is the technique of translating the evaluation of a function that takes multiple arguments into evaluating a sequence of functions, each with a single argument.*

Essentially, with currying we do the following transition:

```
def f(x: Int): Int = x + y
f(1, 2) // evaluates to 3

def curry(f: (Int, Int) => Int): Int => (Int => Int) = x => y => f(x, y)
curry(f) // evaluates to x => (y => x + y)
curry(f)(1) // evaluates to y => y + 1
curry(f)(1)(2) // evaluates to 3
```

Using currying, we can once more improve our `sum` function:

```
def sum(f: Int => Int): (Int, Int) => Int = { // Higher order function
    def sumF(a: Int, b: Int): Int =
        if (a > b) 0
        else f(a) + sumF(a + 1, b)
    sumF // Returns another function
}

sum(cube)(1, 10) // equivalent to sumCubes

// Syntactic sugar:
def sum(f: Int => Int)(a: Int, b: Int): Int =
    if (a > b) 0 else f(a) + sum(f)(a+1, b)
```

Function application associates to the left so `sum(cube)(1, 10)` is equivalent to `(sum(cube))(1, 10)`.

The type of `sum` is `(Int => Int) => (Int, Int) => Int`. This should be read and understood as `(Int => Int) => ((Int, Int) => Int)` as *functional types associate to the right.*

7. Classes: functions and data

In Scala, we use *classes* to define and create data structures:

```scala
class Rational(x: Int, y: Int) {
    def numer = x
    def denom = y
}

val x = new Rational(1, 2)
```

This introduces two entities:

- A new *type* named `Rational`
- A *constructor* `Rational` to create elements of this type

7.1. Methods

One can go further and also package functions operating on a data abstraction into the data abstraction itself. Such functions are called *methods*.

```scala
class Rational(x: Int, y: Int) {
    def numer = x
    def denom = y

    def add(that: Rational) =
        new Rational(
            numer * that.denom + that.numer * denom,
            denom * that.denom)

    override def toString = numer + "/" + denom
}
```

7.1.1. IDENTIFIER The identifier is alphanumeric (starting with a letter, followed by letters or numbers) xor symbolic (starting with a symbol, followed by other symbols). We can mix them by using an alphanumeric name, an underscore _ and then a symbol.

Small practical trick: to define a `neg` function that returns the negation of a `Rational`, we can write:

```scala
class Rational(x: Int, y: Int) {
    ...

    def unary_- : Rational = new Rational(-numer, denom) // space between - and : becau
}
```

The *precedence* of an operator is determined by its first character, in the following priority (from lowest to highest):

- All letters
- `|`
- `^`
- `&`
- `< >`
- `= !`
- `:`
- `+ -`
- `* / %`
- All other symbolic characters

7.1.2. INFIX NOTATION Any method with a parameter can be used like an infix operator:

```
r add s                                r.add(s)
r less s     /* in place of */         r.less(s)
r max s                                r.max(s)
```

7.2. Constructors

Scala naturally executes the code in the class body as an implicit constructor, but there is a way to explicitly define more constructors if necessary:

```scala
class Rational(x: Int, y: Int) {
    def this(x: Int) = this(x, 1)

    def numer = x
    def denom = y
}
```

7.3. Data abstraction

We can improve `Rational` by making it an irreducible fraction using the GCD:

```scala
class Rational(x: Int, y: Int) {
    private def gcd(a: Int, b: Int): Int = if (b == 0) a else gcd(b, a % b)
    val numer = x / gcd(x, y) // Computed only once with a val
    val denom = y / gcd(x, y)

    ...
}
```

There are obviously multiple ways of achieving this; the above code just shows one. The ability to choose different implementations of the data without affecting clients is called *data abstraction*.

7.4. Assert and require

When calling the constructor, using a denominator of 0 will eventually lead to errors. There are two ways of imposing restrictions on the given constructor arguments:

- `require`, which throws an `IllegalArgumentException` if it fails
- `assert`, which throws an `AssertionError` if it fails

This reflects a difference in intent:

- `require` is used to enforce a precondition on the caller of a function
- `assert` is used to check the code of the function itself

```scala
class Rational(x: Int, y: Int) {
    require(y != 0, "denominator must be non-zero")

    val root = sqrt(this)
    assert(root >= 0)
}
```

8. Class Hierarchies

8.1. Abstract classes

Just like in Java, we can have absctract classes and their implementation:

```scala
abstract class IntSet {
    def incl(x: Int): IntSet
    def contains(x: Int): Boolean
}

class Empty extends IntSet { // Empty binary tree
    def contains(x: Int): Boolean = false
    def incl(x: Int): IntSet = new NonEmpty(x, new Empty, new Empty)
}

class NonEmpty(elem: Int, left: IntSet, right: Intset) extends IntSet { // left and r
    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 NonEmpty(elem, left incl x, right)
        if (x > elem) new NonEmpty(elem, left, right incl x)
        else this // already in the tree, nothing to add
}
```

8.1.1. TERMINOLOGY

- `Empty` and `NonEmpty` both *extend* the class `IntSet`
- The definitions of `incl` and `contains` *implement* the abstract functions of `IntSet`
- This implies that the types `Empty` and `NonEmpty` *conform* to the type `IntSet`, and can be used wherever an `IntSet` is required
- `IntSet` is the superclass of `Empty` and `NonEmpty`
- `Empty` and `NonEmpty` are *subclasses* of `IntSet`
- In Scala, any user-defined class extends another class. By default, if no superclass is given, the superclass is `Object`
- The direct or indirect superclasses are called *base classes*

8.1.2. OVERRIDE It is possible to *redefine* an existing, non-abstract definition in a subclass by using `override`.

```scala
abstract class Base {
    def foo = 1
    def bar: Int
}

class Sub extends Base {
    override def foo = 2 // You need to use override
    def bar = 3
}
```

Overriding something that isn't overrideable yields an error.

8.2. Traits

In Scala, a class can only have one superclass. But sometimes we want several supertypes. To do this we can use *traits*. It's declared just like an abstract class, but using the keyword `trait`:

```scala
trait Planar {
    def height: Int // Abstract method as it lacks an implementation
    def width: Int
    def surface = height * width // Concrete method defining a default implementation
}
```

Classes, objects and traits can inherit from at most one class but as arbitrarily many traits.

```
class Square extends Shape with Planar with Movable ...
```

Traits **cannot** have value parameters, only classes can.

8.3. Singleton objects

In the `IntSet` example, one could argue that there really only is a single empty `IntSet`, and that it's overkill to have the user create many instances of `Empty`. Instead we can define a *singleton object*:

```
object Empty extends IntSet {
    def contains(x: Int): Boolean = false
    def incl(x: Int): IntSet = new NonEmpty(x, Empty, Empty)
}
```

Singleton objects are values, so `Empty` evaluates to itself.

8.4. Packages and imports

Classes and objects are organized in packages, just like in Java.

```
package funprog.example

object Rational {
    ...
}
```

One can now call the object using its full qualified name, or with an import:

```
object test {
    new funprog.example.Rational(1, 2)
}

// or
import funprog.example.Rational // Import Rational
import funprog.example.{Rational, Hello} // Import both Rational and Hello
import funprog.example._ // Or import everything in funprog.example

object test2 {
    new Rational(1, 2)
}
```

8.5. Polymorphism

Just like in Java, we may wish to have polymorphic types.

```scala
trait List[T] {
    def isEmpty: Boolean
    def head: T
    def tail: List[T]
}

class Cons[T](val head: T, val tail: List[T]) extends List[T] {
    def isEmpty = false

    // val head: T is a legal implementation of head
    // and so is val tail: List[T]
    // (they're in the argument list of Cons[T])
}

class Nil[T] extends List[T] {
    def isEmpty = true
    def head = throw new NoSuchElementException("Nil.head")
    def tail = throw new NoSuchElementException("Nil.tail") // returns type Nothing
}
```

Type parameters can be used in classes, but also in functions.

8.5.1. TYPE INFERENCE The Scala compiler can usually deduce the correct type parameters.

```scala
def singleton[T](elem: T) = new Cons[T](elem, new Nil[T])

singleton[Int](1) // Explicit type definition
singleton(1) // Type inference
```

8.5.2. TYPE BOUNDS We can set the types of parameters as either subtypes or supertypes of something. For instance, a method that takes an `IntSet` and returns it if all elements are positive, or throws an error if not, could be implemented as such:

```scala
// Can either return an Empty or a NonEmpty, depending on what it's given:
def assertAllPos[S <: IntSet](r: S): S = ...
```

Here, `<: IntSet` is an **upper bound** of the type parameter `S`. Generally:

- `S <: T` means `S` is a *subtype* of `T`
- `S >: T` means `S` is a *supertype* of `T`

It's also possible to mix a lower bound with an upper bound:

```scala
[S >: NonEmpty <: IntSet]
```

This would restrict `S` to any type on the interval between `NonEmpty` and `IntSet`.

8.5.3. VARIANCE Given `NonEmpty <: IntSet`, is `List[NonEmpty] <: List[IntSet]`? Yes!

Types for which this relationship holds are called **covariant** because their subtyping relationship varies with the type parameter. This makes sense in situations fitting the Liskov Substitution Principle (loosely paraphrased):

> *If `A <: B`, then everything one can do with a value of type `B` one should also be able to do with a value of type `A`.*

In Scala, for instance, `Array`s are not covariant.

There are in fact 3 types of variance (given `A <: B`):

- `C[A] <: C[B]` means `C` is **covariant**
- `C[A] >: C[B]` means `C` is **contravariant**

- Neither `C[A]` nor `C[B]` is a subtype of the other means `C` is **nonvariant**

Scala lets you declare the variance of a type by annotating the type parameter:

```scala
class C[+A] { ... } // C is covariant
class C[-A] { ... } // C is contravariant
class C[A] { ... } // C is invariant
```

Functions are contravariant in their argument types, and covariant in their result type. This allows us to state a very useful and important subtyping relation for functions: `A1 => B2 <: A2 => B1` **if and only if** `A1 >: A2` **and** `B1 >: B2`.

Note that, in this case, `A2 => B2` is **unrelated to** `A1 => B1`.

The Scala compiler checks that there are no problematic combinations when compiling a class with variance annotations. Roughly:

- *Covariant* type parameters can only appear in method results
    - However, *covariant* type parameters may appear in *lower* bounds of method type parameters
- *Contravariant* type parameters can only appear in method parameters
    - However, *contravariant* type parameters may appear in *upper* bounds of method type parameters
- *Invariant* type parameters can appear anywhere

The following code, for instance, is correct as the covariant type parameter is a method result, and the contravariant is a parameter:

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

8.6. Object oriented decomposition

Instead of writing external methods that apply to different types of subclasses, we can write the functionality inside the respective classes.

```scala
trait 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
}
```

But this is problematic if we need to add lots of methods but not add many classes, as we'll need to define new methods in all the subclasses. Another limitation of OO decomposition is that some non-local operations cannot be encapsulated in the method of a single object.

In these cases, pattern matching may be a better solution.

8.7. Pattern matching

Pattern matching is a generalization of `switch` from C or Java, to class hierarchies. It's expressed in Scala using the keyword `match`:

```scala
def eval(e: Expr): Int = e match {
    case Number(n) => n
    case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

If none of the cases match, a match error exception is thrown.

Patterns are constructed from:

- Constructors, e.g. `Number`, `Sum`
- Variables, e.g. `n`, `e1`, `e2`
- Wildcard patterns _ (if we don't care about the argument, we can use `Number(_)`)
- Constants, e.g. `1`, `true` (by convention, start `const` with a capital letter).

These patterns can be stacked, so we may try to match a `Sum(Number(1), Var(x))` for instance. The same variable name can only appear once in a pattern, so `Sum(x, x)` is not a legal pattern.

It's possible to define the evaluation function as a method of the base trait:

```scala
trait Expr {
    def eval: Int = this match {
        case Number(n) => n
        case Sum(e1, e2) => e1.eval + e2.eval
    }
}
```

Pattern matching is especially useful when what we do is mainly to add methods (not really changing the class hierarchy). Otherwise, if we mainly create sub-classes, then object-oriented decomposition works best.

8.8. Case classes

A **case class** definition is similar to a normal class definition, except that it is preceded by the modifier `case`. For example:

```scala
trait Epxr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr
```

Doing this implicitly defines companion object with `apply` methods.

```scala
object Number {
    def apply(n: Int) = new Number(n)
}
object Sum {
    def apply(e1: Expr, e2: Expr) = new Sum(e1, e2)
}
```

This way we can just do `Number(1)` instead of `new Number(1)`.

9. Lists

There are two important differences between lists and arrays:

- Lists are immutable — the elements of a list cannot be changed.
- Lists are recursive (linked lists), while arrays are flat.

Like arrays, lists are *homogeneous*: the elements of a list must all have the same type.

9.1. List constructors

A bit of syntactic sugar: you can construct new lists using the construction operation `::` (pronounced *cons*).

```scala
fruit = "apples" :: "oranges" :: "pears" :: Nil
List("apples", "oranges", "pears") // Equivalent
Nil.::("pears").::("oranges").::("apples") // Also equivalent
```

As a convention, operators ending in `:` associate to the right, and are calls on the right-hand operand.

9.2. List patterns

It is also possible to decompose lists with pattern matching. Examples:

```scala
Nil // Nil constant
p :: ps // A pattern that matches a list with a head matching p and a tail matching ps
List(p1, ..., pn) // Same as p1 :: ... :: pn :: Nil
1 :: 2 :: xs // Lists that start with 1 then 2
x :: Nil // Lists of length 1
List(x) // Same as x :: Nil
List() // Empty list, same as Nil
List(2 :: xs) // A list that contains as only element another list that starts with 2

x :: y :: List(xs, ys) :: zs // Lists of length >= 3 with a list of 2 elements in 3rd
```

We can do a really short insertion sort this way (but one that runs in O(n^2))

```scala
def isort(xs: List[Int]): List[Int] = xs match {
    case List() => List()
    case y :: ys => insert(y, isort(ys)) // y is head, ys is tail
}

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)
}
```

9.3. List methods

9.3.1. SUBLISTS AND ELEMENT ACCESS

- `xs.length`: The number of elements of `xs`
- `xs.last`: The list's last elemeent, exception if `xs` is empty
- `xs.init`: A list consisting of all elements of `xs` except the last one, except if `xs` is empty.
- `xs take n`: A list consisting of the first `n` elements of `xs` or `xs` itself if it's shorter than `n`
- `xs drop n`: The rest of the collection after taking `n` elements.
- `xs(n)`: The element of `xs` at index `n`

9.3.2. CREATING NEW LISTS

- `xs ++ ys` or `xs ::: ys`: Concatenation of `xs` and `ys`
- `xs.reverse`: The list containing the elements of `xs` in reversed order
- `xs updated (n, x)`: The list containing the same elements as `xs`, except at index `n` where it contains `x`.

9.3.3. FINDING ELEMENTS

- `xs indexOf x`: The index of the first elemen in `xs` matching `x`, or `-1` if `x` does not appear in `xs`
- `xs contains x`: same as `xs indexOf x >= 0`

9.4. Higher-order list functions

These are functions that work on lists and take another function as argument. The above examples often have similar structures, and we can identify patterns:

- transforming each element in a list in a certain way
- retrieving a list of all elements satisfying a criterion
- combining the elements of a list using an operator

Since Scala is a functional language, we can write generic function that implement these patterns using higher-order functions.

9.4.1. MAP The actual implementation of `map` is a bit more complicated for performance reasons, but follows something allong the lines of:

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

// Multiplies all elements of the list by a factor
def scaleList(xs: List[Double], factor: Double): List[Double] =
    xs map (x => x * factor)

// Squares all elements of the list
def squareList(xs: List[Int]): List[Int] =
    xs map (x => x * x)
```

9.4.2. FILTER

```scala
abstract class List[T] {
    ...
    def filter(p: T => Boolean): List[T] = this match {
        case Nil => this
        case x :: xs =>
            if (p(x)) x :: xs.filter(p)
            else xs.filter(p)
    }
}

def positiveElems(xs: List[Int]): List[Int] = xs filter (x => x > 0)
```

There are a few other methods that extract sublists based on a predicate:

- `xs filterNot p`: Same as `xs filter (x >= !p(x))`
- `xs partition p`: Same as `(xs filter p, xs filterNot p)`
- `xs takeWhile p`: The longest prefix of list `xs` consisting of elements that all satisfy the predicate `p`
- `xs dropWhile p`: The remainder of the list `xs` after any leading elements satisfying `p` have been removed
- `xs span p`: Same as `(xs takeWhile p, xs dropWhile p)`

9.4.3. REDUCE A reduction of a list consist of a combination of the elements using a given operator (i.e. summing or multiplying all the elements).

For certain operations, the order matters, and there are therefore different orders in which the reduction can be made.

One such function is `foldLeft`. It goes from left to right and takes an *accumulator* `z` as an additional parameter, which is returned when `foldLeft` is called on an empty list. For instance

```scala
// The general notation is:
// (List(x1, ..., xn) foldLeft z)(op)
// Which returns:
// ((z op x1) op ...) op xn

def sum(xs: List[Int]) = (xs foldLeft 0)(_ + _)
def product(xs: List[Int]) = (xs foldLeft 1)(_ * _)
```

Note: The `(_ + _)` notation is equivalent to `((x, y) => x + y)`.

`foldLeft` and `reduceLeft` (same as `foldLeft` but without the `z` argument) could be implemented as follows:

```scala
abstract class List[T] {
    ...
    def reduceLeft(op: (T, T) => T): T = this match {
        case Nil     => throw new Error("Nil.reduceLeft")
        case x :: xs => (xs foldLeft x)(op)
    }

    def foldLeft[U](z: U)(op: (U, T) => U): U = this match {
        case Nil     => z
        case x :: xs => (xs foldLeft op(z, x))(op)
    }
}
```

`foldRight` and `reduceRight` follow similar implementations but put the parentheses to the right.

10. Implicit parameters

If we wanted to generalize an implementation of merge sort to work on more types than just `Int`s, we could rewrite it as such:

```scala
def msort[T](xs: List[T])(lt: (T, T) => Boolean): List[T] = {
    val n = xs.length / 2
    if (n == 0) xs
    else {
        def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {
            case (Nil, ys) => ys
            case (xs, Nil) => xs
            case (x :: xs1, y :: ys1) =>
                if (lt(x, y)) x :: merge(xs1, ys)
                else y :: merge(xs, ys1)
        }
        val (fst, snd) = xs splitAt n
        merge(msort(fst)(lt), msort(snd)(lt))
    }
}

val nums = List(2, -4, 5, 6, 1)
msort(nums)((x, y) => x < y)

// Generalisation:
val fruits = List("apple", "pineapple", "orange", "banana")
msort(fruits)((x, y) => x.compareTo(y) < 0) # lexicographical order
```

As a tiny note, it's usually best to put the function value as the last parameter of a function, because that makes it more likely that the compiler can infer the types of the arguments of the function. E.g. we have written `(x, y) => x < y)` instead of `(x: Int, y: Int) => x < y`.

How can we make this code nicer? We can use the `Ordering` type to represent the function, and make it an implicit parameter:

```scala
 1 def msort[T](xs: List[T])(implicit ord: Ordering[T]): List[T] = {
 2     val n = xs.length / 2
 3     if (n == 0) xs
 4     else {
 5         def merge(xs: List[T], ys: List[T]): List[T] = (xs, ys) match {
 6             case (Nil, ys) => ys
 7             case (xs, Nil) => xs
 8             case (x :: xs1, y :: ys1) =>
 9                 if (ord.lt(x, y)) x :: merge(xs1, ys)
10                 else y :: merge(xs, ys1)
11         }
12         val (fst, snd) = xs splitAt n
13         merge(msort(fst), msort(snd)) // ord is visible at this scope
14     }
15 }
16 
17 val nums = List(2, -4, 5, 6, 1)
18 msort(nums)
19 
20 // Generalisation:
21 val fruits = List("apple", "pineapple", "orange", "banana")
22 msort(fruits)
```

Using the `Ordering[T]` type means using the predefined default ordering, which we don't even need to supply to `msort`, namely `Ordering.String` and `Ordering.Int`. See notes on ordering in Java.

When you write an implicit parameter, and you don't write an actual argument that matches that parameter, the compiler will figure out the right implicit to pass, based on the demanded type.

10.1. Rules for implicit parameters

Say that a function takes an implicit parameter of type `T`. The compiler will search for an implicit definition that:

- is marked `implicit`
- has a type compatible with `T`
- is visible at the scope of the function call (see line 13 above), or is defined in a companion object associated with `T`

If there's a single (most specific) definition, it will be taken as actual argument for the implicit parameter. Otherwise, it's an error.

For instance, at line 13, the compiler inserts the `ord` parameter of `msort`

11. Proof techniques

Before we can prove anything, we'll just assert that pure functional languages have a property called *referential transparency*, since they don't have side effects. This means that we can use reduction steps as equalities to some part of a term.

11.1. Structural induction

The principle of structural induction is analogous to natural induction.

To prove a property `P(xs)` for all lists `xs`:

- **Base case**: Show that `P(Nil)` holds
- **Induction step**: for a list `xs`and some element `x`, show that if `P(xs)` holds then `P(x :: xs)` also holds.

Instead of constructing numbers and adding 1, we construct lists from `Nil` and add one element.

11.1.1. EXAMPLE Let's show that, for lists `xs`, `ys` and `zs`, `(xs ++ ys) ++ zs = xs ++ (ys ++ zs)`.

We'll use the two following axioms of ++ to prove this:

1. `Nil ++ ys = ys`
2. `(x :: xs1) ++ ys = x :: (xs1 ++ ys)`

Let's solve it. First, the base case:

```
// Left-hand side:
(Nil ++ ys) ++ zs = ys ++ zs // by the 1st clause of ++

// Right-hand side:
Nil ++ (ys ++ zs) = ys ++ zs // by the 1st clause of ++
```

Now, onto the induction step:

```
// Left-hand side:
((x :: xs) ++ ys) ++ zs = (x :: (xs ++ ys)) ++ zs // by 2nd clause of ++
                        = x :: ((xs ++ ys) ++ zs) // by 2nd clause of ++
                        = x :: (xs ++ (ys ++ zs)) // by induction hypothesis

// Right-hand side:
(x :: xs) ++ (ys ++ zs) = x :: (xs ++ (ys ++ zs)) // by 2nd clause of ++
```

So this property is established.

12. Other collections

All the collections we'll study are *immutable*. The collection hierarchy is as follows:

- `Iterable`
  - `Seq`
    - `List`
    - `Vector`
    - `Range`
  - `Set`
  - `Map`

12.1. Sequences

12.1.1. VECTORS A `Vector` of up to 32 elements is just an array, but once it grows past that bound, its representation changes; it becomes a `Vector` of 32 pointers to `Vectors` (that follow the same rule once they outgrow 32).

Unlike lists, which are linear (access to the end of the list is slower than the start), random access to a certain element in a vector can be done in time $\log_{32}(n)$.

Vectors are fairly good for bulk operations that traverse a sequence, such as a map, fold or filter. Also, 32 is a good number since it corresponds to a cache line.

Vectors are created analogously to lists:

```
val nums = Vector(1, 2, 3, -88)
val people = Vector("Bob", "James", "Peter")

// Instead of x :: xs we have:
x +: xs // create a new vector with leading element x, followed by xs
xs :+ x // create a new vector with trailing element x, preceded by xs
```

Creating new vectors with these `:+` and `+:` operators works by adding a vector, and recreating parent vectors with pointers to the existing ones. Doing this preserves immutability while still being fairly efficient ($\log_{32}(n)$).

12.1.2. ARRAYS AND STRINGS They come from Java, so they can't be subclasses of `Iterable`, but they still work just as if they were subclasses of `Seq`, and we can apply all the same operations.

12.1.3. RANGE Represents a sequence of evenly spaced integers.

```scala
val r: Range = 1 until 5 // 1, 2, 3, 4
val s: Range = 1 to 5 // 1, 2, 3, 4, 5
1 to 10 by 3 // 1, 4, 7, 10
6 to 1 by -2 // 6, 4, 2
```

Ranges are represented as three fields: lower bounds, upper bounds and step value.

12.2. Sets

Sets are another basic abstraction in the Scala collections. It is written analogously to a sequence:

```scala
val fruit = Set("apple", "banana", "pear")
val s = (1 to 6).toSet

// Most operations on sequences are also available on sets:
s map (_ + 2) // Set(3, 4, 5, 6, 7, 8)
fruit filter (_.startsWith == "app") // Set("apple")
s.nonEmpty // true
```

The principal differences between sets and sequences are:

1. Sets are **unordered**: the elements of a set do not have a predefined order in which they appear in the set.
2. Sets **do not have duplicate elements**.
3. The **fundamental operation** on sets is `contains`.

12.3. Maps

Another fundamental collection type is the `map`. A map of type `Map[Key, Value]` is a data structure associating keys with values.

```scala
val romanNumerals = Map("I" -> 1, "V" -> 5, "X" -> 10)
val capitalOfCountry = Map("US" -> "Washington", "Switzerland" -> "Bern")
```

They're both an `Iterable` and a function, as `Map[Key, Value]` also extends the function type `Key => Value`.

```scala
capitalOfCountry("US") // "Washington"
capitalOfCountry("Andorra") // NoSuchElementException: key not found: Andorra

capitalOfCountry get "Andorra" // None
capitalOfCountry get "US" // Some("Washington")
```

Both the `None` and the `Some` are subclasses of the `Option` type.

```scala
trait Option[+A]
case class Some[+A](value: A) extends Option[A]
object None extends Option[Nothing]
```

This means that we can do pattern matching, or use the `withDefaultValue`:

```scala
def showCapital(country: String) = capitalOfCountry.get(country) match {
    case Some(capital) => capital
    case None => "missing data"
}

capitalOfCountry get "US" // "Washington"
capitalOfCountry get "Andorra" // "missing data"

val cap1 = capitalOfCountry withDefaultValue "Unknown"
cap1("Andorra")  // "Unknown"
```

12.4. Operations on iterables

12.4.1. OPERATIONS ON SEQUENCES

- `xs exists p`: `true` if there is an element `x` of `xs` such that `p(x)` holds, `false` otherwise.
- `xs forall p`: `true` if `p(x)` holds for all elements `x` of `xs`, `false` otherwise
- `xs zip ys`: A sequence of pairs drawn from corresponding elements of sequences `xs` and `ys`
- `xs.unzip`: Splits a sequences of pairs `xs` into two sequences consisting of the first and second halves of all pairs
- `xs.flatMap f`: Applies collection-valued function `f` to all elements of `xs` to all elements the results.
- `xs.sum`: The sum of all elements of this numeric collection
- `xs.product`: The product of all elements of this numeric collection
- `xs.max`: The maximum of all elements of this numeric collection (an `Ordering` must exist)
- `xs.min`: The minimum of all elements of this numeric collection (an `Ordering` must exist)

A few examples below.

```scala
// List all combinations of numbers x and y
// where x is drawn from 1..M
// and y is drawn from 1..N
(1 to M) flatMap (x => (1 to N) map (y => (x, y)))
    // > Vector((1, 1), (1, 2), ..., (2, 1), (2, 2), ...)

// Scalar product of two vectors
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
    (xs zip ys).map(xy = xy._1 * xy._2).sum

// Or using pattern matching function value
// Note: Generally, {case p1 => e1 ...} is
// equivalent to x => x match {case p => e1 ...}
def scalarProduct(xs: Vector[Double], ys: Vector[Double]): Double =
    (xs zip ys).map{ case (x, y) => x * y}.sum

def isPrime(n: Int): Boolean =
    (2 until n) forall (d => n % d != 0)
```

12.4.2. SORTED AND GROUPBY

To sort elements, we can use either `sortWith` or `sorted` as below.

`groupBy` is available on Scala collections. It partitions a collection into a map of collections according to a discriminator function `f`.

```scala
val fruit = List("apple", "pear", "orange", "pineapple")
fruit sortWith (_.length < _.length) // List("pear", "apple", "orange", "pineapple")
fruit.sorted // List("apple", "orange", "pear", "pineapple")

fruit groupBy (_.head) // > Map(p -> List(pear, pineapple),
                       // |     a -> List(apple),
                       // |     o -> List(orange))
```

13. For-Expressions

Higher order functions and collections in functional languages often replace loops in imperative languages. Programs using many nested loops can therefore often be replaced by a combination of higher order functions.

For example, let's say we want to find all $1 < i < j < n$ for which $i + j$ is prime. This would take two loops in an imperative language, but in Scala we can "just" write:

```scala
(1 until n).flatMap(i => (1 until i) map (j => (i, j)))
           .filter(pair => isPrime(pair._1 + pair._2))
```

This is hard to read, so we can use a for expression, of the form

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

Where `s` is a sequence of *generators* and *filters*, and `e` is an expression whose value is returned by an iteration.

Instead of `( s )`, braces `{ s }` can also be used, and then the sequence of generators and filters can be written on multiple lines without requiring semicolons.

Using a for expression, we can rewrite our previous example:

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

// Scalar product
(for ((x, y) <- xs zip ys) yield x*y).sum
```

The rest of these notes correspond to the Functional Pogram Design in Scala *course*

13.1. Querying

Let's say we want to query the number of authors who have written two or more books.

```scala
{   for {
        b1 <- books
        b2 <- books
        if b1.title < b2.title // Prevent duplicates by using lexicographical order
                               // We could also use if b1 != b2, but this would
                               // match for the same pair of books twice.
        a1 <- b1.authors
        a2 <- b2.authors
        if a1 == a2
    } yield a1
}.distinct // another way to prevent duplicates
```

The first mechanism to prevent duplicates is to compare titles using lexicographical order instead of a simple `!=`. Another trick is to use `.distinct`, which is like a `.toSet`.

13.2. Translation to higher-order functions

The syntax of for is closely related to the higher-order functions `map`, `flatMap`, and `filter`. These functions could be implemented as such:

```scala
def map[T, U](xs: List[T], f: T => U): List[U] =
    for (x <- xs) yield f(x)

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

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

In reality, the translation is done the other way by the compiler. How do we translate for-expressions to these higher-order functions?

Below is the for expression and its translation at the next line.

```scala
// For-expression
for (x <- e1) yield e2
// Desugared
e1.map(x => e2)


// Let s be a (potentially empty) sequence of generators and filters
// For-expression
for (x <- e1 if f; s) yield e2
// Desugared
for (x <- e1.withFilter(x => f); s) yield e2

// For-expression
for (x <- e1; y <- e2; s) yield e3
// Desugared
e1.flatMap(x => for (y <- e2; s) yield e3)

// For-expression
for {
    i <- 1 until n
    j <- 1 until i
    if isPrime(i + j)
} yield (i, j)
// Desugared
(1 until n) flatMap(i =>
    (1 until i).withFilter(j => isPrime(i + j))
               .map(j => (i, j)))
```

See more examples of desugared for-expressions in this gist.

Interestingly, the translation of `for` is not limited to lists, sequences, or collections. Since it's based solely on the presence of the methods `map`, `flatMap` and `withFilter`, we can simply redefine these methods for our own types.

If, for instance, we were to write a database supporting these methods, then as long as these methods are defined, we can use the `for` syntax for querying the database.

13.3. Functional Random Generators

13.3.1. DEFINITION We could also define these three methods (`map`, `flatMap`, `withFilter`) for a random value generator. Let's define it as such:

```scala
trait Generator[+T] {
    def generate: T
}

val integers = new Generator[Int] {
    val rand = new java.util.Random
    def generate = rand.nextInt()
}

val booleans = new Generator[Boolean] {
    def generate = integers.generate > 0
}

val pairs = new Generator[(Int, Int)] {
    def generate = (integers.generate, integers.generate)
}
```

But we can streamline this:

```scala
val booleans = for (x <- integers) yield x > 0

def pairs[T, U](t: Generator[T], u: Generator[U]) = for {
    x <- t
    y <- u
} yield (x, y)
```

Which expands to:

```scala
val booleans = integers map (x => x > 0)

def pairs[T, U](t: Generator[T], u: Generator[U]) =
    t flatMap (x => u map (y => (x, y)))
```

We therefore need to define `map` and `flatMap` on the `Generator` class.

```scala
trait Generator[+T] {
    self => // an alias for "this"
    def generate: T

    def map[S](f: T => S): Generator[S] = new Generator[S] {
        def generate = f(self.generate) // we use self instead of this to reference th
    }

    def flatMap[S](f: T => Generator[S]): Generator[S] = new Generator[S] {
        def generate = f(self.generate).generate
    }
}
```

Our example now expands to:

```scala
val booleans = for (x <- integers) yield x > 0
val booleans = integers map { x => x > 0}
val booleans = new Generator[Boolean] {
    def generate = (x: Int => x > 0)(integers.generate)
}
val booleans = new Generator[Boolean] {
    def generate = integers.generate > 0
}
```

We can also define other types of generators:

```scala
def single[T](x: T): Generator[T] = new Generator[T] {
    def generate = x // identity
}

def choose(lo: Int, hi: Int): Generator[Int] =
    for (x <- integers) yield lo + x % (hi - lo)

def oneOf[T](xs: T*): Generator[T] = // T* means you can give it as many arguments as y
    for (idx <- choose(0, xs.length)) yield xs(idx)
```

13.3.2. USAGE Having created a generator, we can use this as a building block for more complex expressions:

```scala
def lists: Generator[List[Int]] = for {
    isEmpty <- booleans
    list <- if (isEmpty) emptyLists else nonEmptyLists
} yield list

def emptyListst = single(Nil)
def nonEmptyLists = for {
    head <- integers
    tail <- lists
} yield head :: tail
```

13.3.3. APPLICATION: RANDOM TESTING Generators are especially useful for random testing. Obviously it's hard to predict the result of any random input without running the program, but what we can do is test *postconditions*, which are properties of the expected result.

```scala
def test[T](g: Generator[T], numTimes: Int = 100)(test T => Boolean): Unit = {
    for (i <- 0 until numTimes) {
        val value = g.generate
        assert(test(value), "test failed for "+value)
    }
    println("Passed " + numTimes + " tests")
}
```

We can use a tool called ScalaCheck to do this in a more automated way. Instead of writing tests, with ScalaCheck we write *properties* that are assumed to hold. ScalaCheck will then try to find good counter-examples if the assertion fails.

```scala
forall { (l1: List[Int], l2: List[Int]) =>
    l1.size + l2.size == (l1 ++ l2).size
}
```

14. Monads

14.1. Definition

A monad `M` is a parametric type `M[T]` with two operations, `unit` and `flatMap` (more commonly called `bind` in the literature):

```scala
trait M[T] {
    def flatMap[U](f: T => M[U]): M[U]
    def unit[T](x: T): M[T]
}
```

The unit method return a monad with the given type:

- `List` is a monad with `unit(x) = List(x)`
- `Set` is a monad with `unit(x) = Set(x)`
- `Option` is a monad with `unit(x) = Some(x)`

- `Generator` is a monad with `unit(x) = single(x)`

For every monad, `map` can be be defined as a combination of `flatMap` and `unit`. All of the following are equivalent.

```
m map f
m flatMap (x => unit(f(x)))
m flatMap (f andThen unit)
```

These methods have to satisfy some laws:

- **Associativity**: we can put the parentheses either to the left or the right, so `(m flatMap f) flatMap g == m flatMap(x => f(x) flatMap g)`
- **Left unit**: `unit(x) flatMap f == f(x)`
- **Right unit**: `m flatMap unit == m`

14.2. Significance of the laws

Associativity says that one can "inline" nested for-expressions; the following are equivalent:

```
for {
    y <- for(x <- m; y <- f(x)) yield y
    z <- g(y)
} yield z

for {
    x <- m
    y <- f(x)
    z <- g(y)
} yield z
```

Right unit says `for (x <- m) yield x` is equivalent to just `m`, and left unit isn't very useful for for-expressions.

If monads are still mysterious, this is a good read.

15. Streams

Sometimes, for performance reasons, we want avoid computing the tail of a sequence until it is needed for the evalutation result (which might be never). Streams implement this idea while keeping the notation concise. They're similar to lists, but their tail is evaluated only on demand.

15.1. Definition

Streams can be constructed like most other collections:

```
Stream.cons(1, Stream.cons(2, Stream.empty))
Stream(1, 2, 3)
(1 to 1000).toStream
```

`.toStream` can be applied to any collection.

Streams can be described as partially constructed lists, and *they support almost all of the `List` methods*. For instance, to find the second prime number between 1000 and 10000, we can do:

```
((1000 to 10000).toStream filter isPrime)(1)
```

The only exception is the cons operator, which is `#::` instead of `::`. This can be used in operation but also in patterns.

15.2. Implementation

Again, this is pretty close to lists:

```scala
trait Stream[+A] extends Seq[A] {
    def isEmpty: Boolean
    def head: A
    def tail: Stream[A]
    ...
}
```

All other methods can be defined in terms of these three. The actual implementation of streams is in the `Stream` companion object, so if we want to define a new type of `Stream`, we just need to redefine these three methods.

```scala
object Stream {
    def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] { // Use CBN!
        def isEmpty = false
        def head = hd
        def tail = tl
    }
    val empty = new Stream[Nothing] {
        def isEmpty = true
        def head = throw new NoSuchElementException("empty.head")
        def tail = throw new NoSuchElementException("empty.tail")
    }

}
```

Notice how the `cons` method uses CBN. This is what makes the whole drastic difference between `List` and `Stream`!

The other stream methods are implemented analogously to their list counterparts:

```scala
class Stream[+T] {
    ...
    def filter(p: T => Boolean): Stream[T] =
        if (isEmpty) this
        else if (p(head)) cons(head, tail.filter(p))
        else tail.filter(p)
}
```

15.3. Lazy Evaluation

The proposed implementation suffers from a serious potential performance problem: if `tail` is called several times, the corresponding stream will be recomputed each time. To avoid this, we can store the result of the first evalutation of `tail` and re-use the stored result next time.

This is called *lazy evaluation* (as opposed to *by-name evaluation* where everything is recomputed, and *strict evaluation* for normal parameters and `val` definitions). Scala uses strict evaluation by default, but allows lazy evaluation:

```scala
lazy val x = expr
```

`x` is computed only once, when it is needed the first time; since functional programming expressions yield the same result on each call, the result is saved and reused next time.

This means that using a lazy value for `tail`, `Stream.cons` can be implemented more efficiently:

```scala
def cons[T](hd: T, tl: => Stream[T]) = new Stream[T] {
    def head = hd
    lazy val tail = tl
    ...
}
```

15.4. Infinite Streams

Infinite streams benefit from laziness. All elements of a stream except the first one are computed only when needed. This opens up the possibility to define infinite streams.

```scala
def from(n: Int): Stream[Int] = n #:: from(n+1)
val nats = from(1) // stream of all natural numbers
nats map(_ * 4) // all natural multiples of 4
```

We also don't need to worry too much about infinite recursions with infinite streams since the tail isn't evaluated:

```scala
def sqrtStream(x: Double): Stream[Double] = {
    def improve(guess: Double) = (guess + x / guess) / 2
    lazy val guesses: Stream[Double] = 1 #:: (guesses map improve)

    guesses
}

def isGoodEnough(guess: Double, x: Double) =
    math.abs((guess * guess - x) / x) < 0.0001

sqrtStream(4) filter (isGoodEnough(_, 4))
```

16. Functions and State

So far we've seen that rewriting can be done anywhere in a term, and all rewritings which terminate lead to the same solution. For instance:

```scala
def iterate(n: Int, f: Int => Int, x: Int) =
    if (n == 0) x
    else iterate(n-1, f, f(x))
def square = x * x

iterate(1, square, 3)
// Can be rewritten as follows:
if (1 == 0) 3 else iterate(1-1, square, square(3))
iterate(0, square, square(3))
iterate(0, square, 3*3)
iterate(0, square, 9)
if (0 == 0) 9 else iterate(0-1, square, 9)
9

// But also:
if (1 == 0) 3 else iterate(1-1, square, square(3))
iterate(0, square, square(3))
if (0 == 0) square(3) else iterate(0-1, square, square(square(3)))
square(3)
9
```

There are multiple ways to rewrite our way to the solution; this is known as the Church-Rosser Theorem of lambda-calculus.

In this chapter, we'll look at code that *doesn't* satisfy that property. We will say goodbye to the substitution model for code that isn't purely functional.

16.1. Stateful Objects

An object *has a state* if its behavior is influenced by its history. It is mutable (while everything so far has been immutable).

Mutable states are defined using the `var` keyword (instead of `val`), and assigned with `=`:

```
var x: String = "abc"
var count = 111
x = "hi"
count = count + 1
```

If we define an object with stateful variables, then it is a stateful object if the result of calling a method depends on the history of the called methods, that the result may change over time.

16.2. Identity

Mutable state introduces questions about equality, identity between two objects.

With immutable values (`val`), we had *referential transparency*; `val x = E; val y = E` was equivalent to `val x = E; val y = x`. This is no longer the case.

If `BankAccount` is a stateful object (its balance may change), then `val x = new BankAccount` and `val y = new BankAccount` aren't equal. This makes sense, because modifying `x` doesn't mean modifying `y`, and we therefore have to different accounts.

In general, to determine equality, we must first specify what is meant by "being the same". The precise meaning is defined by the property of *operational equivalence*: informally, `x` and `y` are operationally equivalent if *no possible test* can distinguish between them. For any arbitrary function `f`, `f(x, y)` and `f(x, x)` must return the same value.

16.3. Loops

```
// While:
while (i > 0) {
    ...
}

// Do-while:
do {
    ...
} while (i <= 25)

// For:
for (i <- 1 until 3) { // i takes values 1, 2 but not 3
    ...
}
```

For-loops look similar to for-expressions, but are translated to `foreach` instead of `map` and `flatMap`:

```
for (i <- 1 until 3; j <- "abc") print(i + "" + j + " ")
// translates to:
(1 until 3) foreach (i => "abc" foreach (j => print(i + "" + j + " ")))
```

This should print "1a 1b 1c 2a 2b 2c"

17. Lisp

I don't have a whole lot of notes on this, since most of Lisp was seen during lab sessions, and my notes on lambda-calculus are on paper (it wouldn't have been easy typing it in real time). But for future reference, I'm adding a syntax list of the Lisp dialect seen in class:

- `(if c a b)`: special form which evaluates `c`, and then `a` if `c != 0` and `b` if `c = 0`.
- `(cond (c1 r1) ... (cn rn) (else relse))`: special form which evaluates `c1`, then `r1` if `c1` is true, or else continues with the other clauses.
- `(cons first rest)`: constructs a list equivalent to Scala's `x :: xs`. In our interpreter, `xs` must be a list.
- `(car lst)`: returns the head of a given list.
- `(cdr lst)`: returns the tail of a given list
- `(quote x)`: returns x as a quoted expression, i.e. `(quote foo)` returns the quoted symbol `foo`, and `(quote (a b c))` returns the list equivalent to `(cons (quote a) (cons (quote b) (cons (quote c) nil)))`
- `(= a b)`: returns whether `a` and `b` are equal. In our interpreter, a and b may be numbers, symbols or even lists.
- `(lambda (p1 ... pn) body)`: creates an anonymous function.
- `def f x`: creates a definition.
- `def (f p1 ... pn) body`: syntactic sugar for defining a named function.