ADVANCED PROGRAMMING LANGUAGES
Fall 2024, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

Functional programming in Scala

Scala was created and developed by Martin Odersky at the Ecole Polytechnique Federale de Lausanne (EPFL).

It was officially released for the Java platform in early 2004 and for .Net framework in June 2004. Later on, Scala dropped .Net support in 2012. The name Scala is a combination of *scalable* and *language*, signifying that it is designed to grow with the demands of its users.

Fully interoperable with Java, Scala is a **strong, statically typed, general-purpose** programming language that integrates **object-oriented and functional programming** within a very elegant framework. The design of Scala is the result of the brilliant engineering of decades of foundational research work on programming languages and typing systems that has given birth functional languages such ML, Scheme, and Haskell, as well as object-oriented languages such a Smalltalk, F#, and of course Java.

Today Scala source can be **compiled to Java bytecode** and and run on a Java virtual Machine (JVM), on which it provides **full interoperability with Java**, so that libraries written in either language may be referenced directly in Scala or Java code. Unlike Java, Scala has a clean syntax: since Scala 3, there is also an option to use indenting (a.k.a. the *off-side rule*) to structure block which minimizes the heavy boilerplate code that makes Java overly verbose.

Distinctive of Scala are

- **support for** the trademark features of functional programming: immutability, **higher-order polymorphic functions**, **pattern matching**, **lazy evaluation**;
- expressive **strong, static typing** system with rich **algebraic data types**, **parametric types** with **variance annotations**, **type classes**;
- provision for **code reuse** mechanisms based on traits and mixins that extend Java's classes and interfaces
- rich wealth of **libraries**, powerful tools and API available to create applications in a wide range target domains.

While it provides extensive support for functional programming, Scala is **not a *pure* functional language**, in that it hosts variables, assignment and other side-effecting constructs. Having said that,

in our first overview, below, and throughout the course we will leave the imperative features on the background and focus on Scala's pure subset.

REPL

A Scala program is a set of value, type and function definitions, which may grouped into packages. Program execution amounts to evaluating an expression build out of the definitions.

The simplest interaction with Scala is with the interactive Interpreter: REPL - Read-Eval-Print-Loop

- REPL reads defs and expressions, checks and infers types, and if no type error is found executes and prints the result.
- The type system does not allow casts or other loopholes

```
scala> 17 * 3
val res0: Int = 51
```

Bindings : associate identifiers to expressions and values

```
scala>  val x = 13 * 2 + 5
val x: Int = 31
```

Note that the type checker discerns the type of the value from its assignment, a process known as *type inference*. Values defined without a type are not type-less; they are assigned the proper type just as if the type had been included in the definition.

Although type inference will deduce the correct type to use to store data, it will not override an explicit type that you set. If you define a value with a type that is incompatible with the initial value you will get an error:

```
scala> val x: Int = "Hello"
-- [E007] Type Mismatch Error: ------------------------------------------------
1 |val x:Int = "Hello"
  |            ^^^^^^^
  |            Found:    ("Hello" : String)
  |            Required: Int.
```

Identifiers introduced with a `val` are immutable: they can be assigned data when defined, but can never be reassigned (beware: in REPL, new `val` definitions for the same identifier hide previous definitions. In fact, each new definition in REPL opens a new scope)

Expression blocks and local bindings. In its pure subset, each piece of Scala code is an expression. Multiple expressions can be combined using curly braces (`{` and `}`) to create a single *expression*

block. An expression has its own scope, and may contain values and variables local to the block: the last expression in the block is the return value for the entire block.

```
scala> val amount = {  val x = 5 * 20;   x + 10 }
val amount: Int = 110
```

The `x` identifier is defined locally to the block: the scope of this definition is the block. Expression blocks can span multiple lines, and use indentation (recommended) instead of curly brackets.

```
scala> val amount =
     |   val x = 5 * 20
     |   x + 10
val amount: Int = 110
```

Primitive types

Booleans. true, false, operators (==, &&, ||, !)

```
true, false : Boolean
```

Integers.

```
0, 1, 2, -5, : Int
```

Strings.

```
"Hello World" : String
```

Doubles.

```
1.2, 3.14159, 2 : Double
```

There isn't much else to say about primitive types, but one thing is worth mentioning about operators.

In Scala all operators are methods. In fact the Scala design is centered around the rather radical choice that, implementation-wise, everything is an object.

Therefore, unlike Java, where each primitive types comes in two versions, *boxed* vs *unboxed*, in Scala all primitive types are objects, and their operators methods. Luckily, the language supports all the necessary machinery to preserve the familiar look and feel of primitive types we are used to. To illustrate, the syntactic form `2+3` is translated into the following method invocation before being

executed `2.+(3)` : this syntax (which is perfectly legal Scala Syntax) makes it explicit that what is being executed is the invocation of method `+` on object `2` with parameter `3`. The actual nature of `Int` as objects becomes similarly apparent when we explicitly invoke methods as in `2.toString`.

Tuples.

```
scala> val triple = (4, 5, “Griffendor”)
val triple: (Int, Int, String) = (4,5,Griffendor)
```

As we can see, tuples are generated very simply by assembling its elements, which can have different types. Tuples can have an arbitrary number of components and each component can be accessed by a corresponding selector: if the tuple has `n` components, it correspondingly has `n` selectors noted `_i` for all `i` in the range `1..n`. Thus:

```
triple._1 = 4,
triple._2 = 5,
triple._3 = Griffendor
```

Question: what would you think happens if we try `triple._4` ?

Answer: The type checker complains (luckily!)

```
scala> triple._4
-- [E008] Not Found Error: ----------------------------------------------------
1 |triple._4
  |^^^^^^^^^
  |value _4 is not a member of (Int, Int, String) - did you mean triple._1?
1 error found
```

LISTS

Lists are predefined in Scala, and can be constructed very conveniently with the `List` constructor:

```
scala> val listofthree = List(1, 2, 3)
listOfThree : List[Int] = List(1, 2, 3)
```

The actual constructors are `::` (*cons*, right associative) and `Nil` so that `List(1,2,3)` is short for `1::2::3::Nil`. Lists have two main selectors `List.head` and `List.tail` and a very rich set of operators.

What it is	What it does
List() or Nil	The empty List
List(“Cool”, “tools”, “rule”)	Creates a new List[String] with the three values “Cool”, “tools”, and “rule”
val thrill = “Will” :: “fill” :: “until” :: Nil	Creates a new List[String] with the three values “Will”, “fill”, and “until”
List(“a”, “b”) ::: List(“c”, “d”)	Concatenates two lists (returns a new List[String] with values “a”, “b”, “c”, and “d”)
thrill(2)	Returns the element at index 2 (zero based) of the thrill list (returns “until”)
thrill.count(s => s.length == 4)	Counts the number of string elements in thrill that have length 4 (returns 2)
thrill.drop(2)	Returns the thrill list without its first 2 elements (returns List(“until”))
thrill.dropRight(2)	Returns the thrill list without its rightmost 2 elements (returns List(“Will”))
thrill.exists(s => s == “until”)	Determines whether a string element exists in thrill that has the value “until” (returns true)
thrill.filter(s => s.length == 4)	Returns a list of all elements, in order, of the thrill list that have length 4 (returns List(“Will”, “fill”))
thrill.forall(s => s.endsWith(“l”))	Indicates whether all elements in the thrill list end with the letter “l” (returns true)
thrill.foreach(print)	Executes the print statement on each of the strings in the thrill list (prints “Willfilluntil”)
thrill.head	Returns the first element in the thrill list (returns “Will”)
thrill.init	Returns a list of all but the last element in the thrill list (returns List(“Will”, “fill”))
thrill.isEmpty	Indicates whether the thrill list is empty (returns false)
thrill.last	Returns the last element in the thrill list (returns “until”)
thrill.length	Returns the number of elements in the thrill list (returns 3)

What it is	What it does
thrill.map(s => s + “y”)	Returns a list resulting from adding a “y” to each string element in the thrill list (returns List(“Willy”, “filly”, “untily”))
thrill.mkString(", ")	Makes a string with the elements of the list (returns “Will, fill, until”)
thrill.remove(s => s.length == 4)	Returns a list of all elements, in order, of the thrill list *except those* that have length 4 (returns List(“until”))
thrill.reverse	Returns a list containing all elements of the thrill list in reverse order (returns List(“until”, “fill”, “Will”))

Like tuples, lists are immutable in Scala. Unlike tuples, lists elements must all have the same type. Beware of subtyping, however!

```
scala> val ml = List(1, "aaa", List(1,'c'), 2.3)
val ml : List[Matchable] = List(1, aaa, List(1, c), 2.3)
```

Conditional Expressions

Assume `x` and `y` are two `Int` values. The following expression calculates the maximum between them

```
scala> val max = if (x > y) x else y
max: Int = 20
```

Both branches of the conditional must have the same type, which is the type of the conditional expression. Conditionals may span more than one line, in which case expression blocks for the two branches may be used to improve readability.

Conditionals may also come without the `else` branch, in which case the return type is `Unit`, the type of no values that the impure fragment of Scala employs to account for statements.

Match Expressions

In their simplest form `match` expressions are akin to C’s and Java’s “switch” statements, where a single input item is evaluated and the first pattern that is “matched” is executed and its value returned.

Like C's and Java's "switch" statements, Scala's match expressions support a default or wildcard "catch-all" pattern. Unlike them, **only zero or one pattern can match; there is no "fall-through" from one pattern to the next one** in line, nor is there a "break" statement that would prevent this fall-through.

The traditional "switch" statement is limited to matching by value, but Scala's match expressions are an amazingly flexible device that also enables matching such diverse items as types, regular expressions, numeric ranges, and data structure contents. Here we look at the basic uses of match expressions: we'll return to them when discussing algebraic data types.

Here's a definition of `max` using match instead of conditional expressions:

```
val max = x > y match
     |   case true => x
     |   case false => y
```

Here's a code snipped testing the `status` value returned by a server in response to a service request:

```
val message = status match
     |     case 200 =>
     |         "ok"
     |     case 400 =>
     |         "ERROR - we called the service incorrectly"
     |     case 500 =>
     |         "ERROR - the service encountered an error"
```

Expression Loops (a.k.a. *for comprehensions*)

A *loop expression* is a term for exercising a task repeatedly, iterating over a range of data executing an expression every time and **return a collection of all the expression's return values**. Scala for-loops are highly customizable, supporting nested iterating, filtering, and value binding.

We illustrate for loops at work with `Ranges` created using the `to` operator.

```
scala> for (x <- 1 to 7) yield  "Day "++ x.toString
val res0: IndexedSeq[String] = Vector(Day 1, Day 2, Day 3, Day 4, Day 5, Day 6, Day 7)
```

Note that the result has type `IndexedSeq[String]`, an indexed sequence of `String`, and is assigned a `Vector`, one of the implementations of `IndexedSeq`. Because of Scala's support for subtyping, a `Vector` (a subtype of `IndexedSeq`) can be assigned to an `IndexedSeq`-typed value.

For loops may be used with *filtering guards*:

```
scala> val threes = for (i <- 1 to 20 if i % 3 == 0) yield i
threes: IndexedSeq[Int] = Vector(3, 6, 9, 12, 15, 18)
```

Functions

At its most basics, a Scala function is a named wrapper for an expression. Here is an example of defining and invoking input-less functions:

```
scala> def hi : String = "hi"
hi: String
scala> hi
res0: String = "hi"
```

As with values, the return type of functions may be inferred, but just as like values functions are easier to read with explicit types.

More interestingly than the previous example, functions can take parameters:

```
scala> def max(x : Int, y : Int) : Int =
    if (x > y) then x else y
def max(x: Int, y: Int): Int
```

Recursive and tail recursive functions

Recursive functions are functions that invoke themselves, typically with some type of parameter or external condition that will be checked to avoid an infinite loop of function invocation. Here's an example of a recursive function that raises an integer by a given positive exponent:

```
scala> def power(x: Int, n: Int): Int =
     |   if (n >= 1) x * power(x, n-1)
     |   else 1
power: (x: Int, n: Int) : Int
```

Recursive functions are elegant, but often inefficient, as their execution uses stack space (and may in fact cause the dreaded "Stack Overflow" error, where invoking a recursive function too many times eventually using up all of the available stack space).

To prevent that, the Scala compiler can optimize recursive functions with *tail recursion*, a technique that can be activated when the recursive call is the last expression to be evaluated in the function body, and hence may reuse the existing function stack space. Only functions whose last expression is the recursive invocation can be optimized for tail-recursion: in, instead, the result of invoking itself is used for anything but the direct return value, a function can't be optimized.

To mark a function as intended for tail-recursion, Scala provides the annotation `@annotation.tailrec` before the function definition or on the previous line. Here's the same example again only marked with the `tailrec` annotation, to let Scala know we expect it to be optimized for tail-recursion: if it cannot be, the compiler should treat it as an error:

```
scala> @annotation.tailrec
          def power(x: Int, n: Int) : Int  =
               if (n >= 1) x * power(x, n-1) else 1
-- Error: ----------------------------------------------------------------------
3 |     if (n >= 1) x * power(x, n-1) else 1
  |                     ^^^^^^^^^^^^
  |                Cannot rewrite recursive call: it is not in tail position
```

Uh, the function couldn't be optimized because the recursive call is not the last statement in the function. This is understandable. Let's try to switch the "if" and "else" conditions and try again:

```
scala> @annotation.tailrec
          def power(x: Int, n: Int) : Int  =
               if (n < 1) 1 else x * power(x, n-1)
-- Error: ----------------------------------------------------------------------
3 |     if (n < 1) 1 else x * power(x, n-1)
  |                          ^^^^^^^^^^^^
  |                          Cannot rewrite recursive call: it is not in tail positic
```

Same problem: now the recursive call *looks* like it's the last expression, but if we consider how the expression is evaluated we see that we are taking the result of the recursive call and *then* multiplying it by a value, so that the last operation is the multiplication, not the recursive call.

The standard way to fix this is to move the multiplication into the invoked function instead of multiplying its result: that's the familiar tail recursion technique based on accumulators.

```
scala> def power(x : Int, n : Int) : Int =
           @annotation.tailrec
           def tr_power(x : Int, n : Int, acc : Int) : Int =
                  if (n >= 1) tr_power(x, n-1, acc*x) else acc
           tr_power(x,n,1)
```

Higher Order Functions

All the functions discussed so far have been introduced with the `def` keyword. But functions are ordinary values in Scala, hence they can be bound to identifiers - and similarly, stored in data structures or passed as arguments to other functions - as any other value. We start with a simple example:

```
scala> val abs: (Int => Int) = (n : Int) => if (n < 0) then -n else n
val abs: Int => Int = Lambda$1343/1460346967@7cd3e0da
```

Here `Int => Int` is the type of functions that, when applied to an integer value return an integer (or raise and exception, but we'll ignore that, for the time being). The body of the definition:

```
(n : Int) => if (n < 0) then -n else n
```

is a *function literal*, i.e. an expression that represents the function mapping its argument to the corresponding absolute value.

Binding functions to names is not very surprising. A more interesting, and paradigmatic, practice in functional programming is the case of *higher-order functions*, i.e. functions that accept other functions as arguments, a mechanism that provides a very powerful form of abstraction in program design. Later on along the course, we will see how effective this mechanism is, and how it constitutes one of the core features of the functional programming style. Here, we illustrate it with a simple example (from [FP-Scala])

The following function builds a formatted string to document the result of an application of the `abs` function defined earlier:

```
scala> def formatAbs(n: Int) =
             val msg = "The absolute value of %d is %d"
             msg.format(n, abs(n))
def formatAbs(n: Int): String
```

The body of the function uses `format`, a (rather convenient) predefined method of type `String`: in the case in question, it returns a copy of the string `msg` on which it is invoked with the values of the arguments `n` and `abs(n)` substituted for the occurrences of `%d`.

The same formatting exercise can be tried with the factorial function we defined in our previous lecture:

```
scala> def formatFact(n: Int) =
             val  msg = "The factorial of %d is %d"
```

```
            msg.format(n, fact(n))
def formatFact(n: Int): String
```

As we can see, the two formatting functions are almost identical. In fact, we can identify the common patterns and represent them uniformly as function parameter that abstracts over the two and can be instantiated so as to capture their behavior.

```
scala> def format(n: Int, name : String, f: Int => Int) =
            val  msg = "The %s of %d is %d"
            msg.format(name, n, f(n))
def format(n: Int,name : String, f: Int => Int): String
```

`format` is an example of higher-order function whose last parameter is a function. Now `formatAbs` and `formatFact` can be obtained as instances of `format`:

```
def formatAbs(n : Int) = format(n, "abs", abs)
def formatFact(n : Int) = format(n, "fact", fact)
```

Clearly, now that we have `format` we may use it to format the result of any `Int => Int` function (see the `formatCube` exercise below).

Polymorphic Functions

So far we have only seen examples of monomorphic functions that operate just on one type. Often, the power of higher-orderness is amplified by having functions operate parametrically on types. Again, this is a feature that we will discuss in detailed in the coming lectures, but it's good to introduce the idea right away.

The following function returns the first index at which a given string `key` occurs in an array of strings. ,

```
def findFirst(strings: Array[String], key: String): Int =
  @annotation.tailrec
  def loop(n: Int): Int =
    if n >= strings.length then -1
    else if strings(n) == key then n
         // `strings (n)` extracts the n'th element
    else loop(n + 1)
  loop(0)
```

The details are not that important here. What's important is that the code for this function would look identical if we were searching an `Int` in an `Array[Int]`, or more generally an `A` on any `Array[A]`. In this latter case, however, we need to abstract away from the equality test used above:

that can be done by making the function parametric in a further parameter representing a tester function:

```
def polyFindFirst[A](as: Array[A], key: A, p: A => Boolean): Int =
   @annotation.tailrec
   def loop(n: Int): Int =
      if n >= as.length then -1
      else if p(as(n)) then n
      else loop(n + 1)
   loop(0)
```

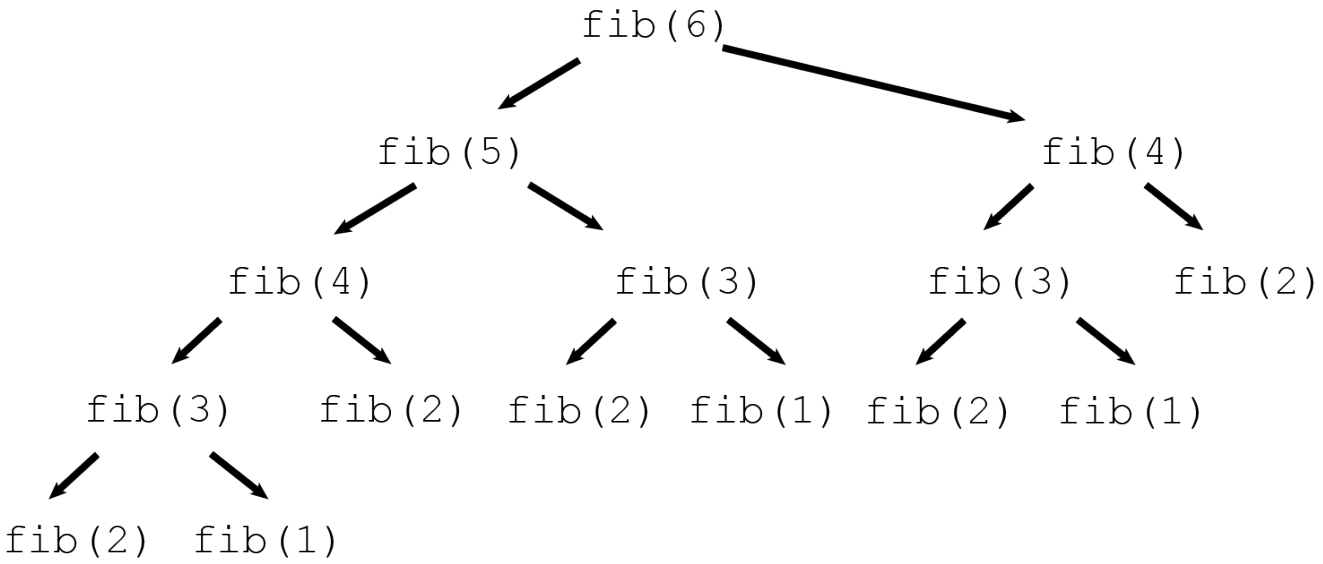
This is an example of polymorphic, or *generic* (higher-order) function. We are abstracting over the type of the array and the function used to search the array. The type parameter `A` can be referenced in the body of the function (as well as in the parameter list as it happens in this case).

Exercises

1. Tail recursive Fibonacci Write a tail recursive version of the Fibonacci function.
Note: The mathematical definition of the Fibonacci numeral sequence is as follows:

$$\text{fib}(0) = 0$$
$$\text{fib}(1) = 1$$
$$\text{fib}(n) = \text{fib}(n\text{-}1) + \text{fib}(n\text{-}2)$$

If we look at the call tree (below), we easily see its exponential growth.



The tail recursive definition is one way to avoid this exponential complexity.
Hint: use two accumulators for the values of fib (n) and fib(n+1)

2. Count change Given a list of coin denominations, write a recursive function `count_change(coins: List[Int], amnt: Int): Int` that counts the number of unique (up to re-orderings of the coins used) ways to make change.

Examples:

- count_change(List(1,5,10,25), 11) = 4 because
  11 = 1+1+1+1+1+1+1+1+1+1+1 = 1+1+1+1+1+1+5 = 1+5+5 = 1+10
- count_change(List(List(5, 2, 3), 10) = 4 because
  10 = 5+5 = 2+3+5 = 2+2+2+2+2 = 2+2+3+3
- count_change(List(List(1, 2), 4) = 3
  4 = 1+1+1+1 = 1+1+2 = 2+2

Hint. For the base case, there are two possibilities:

- If the target amount is 0, there is only one way to make the sum, which is by not selecting any coin, so the count is 1.
- If the target sum amount is negative or no coins are left to consider , then there are no ways to make the amount, so the count is 0

Otherwise, chosen any coin in the list, there are 2 options.

- *include the coin*: Subtract the coin's denomination from the target sum and call the count function recursively with the updated sum and the same list of coins
- *exclude the coin*: Call the count function recursively with the same sum and the remaining coins.
  The final result will be the sum of both cases.]

3. More formatting Use the higher-order `format` function defined earlier to define `format_cube : Int => String` , to format the *cube* function (i.e. the function that given an `n` returns `n` to the power of 3).

4. Find first occurrence of string: Define the `findFirst` function as an instance of `polyFindFirst` operating on strings.

5. Check that list is sorted We want a function `sorted[A](as:List[A], lte:(A,A)=>Boolean): Boolean` which checks whether `as: List[A]` is sorted according to a given comparison function `lte` (the latter returns true if its first argument is less than or equal to its second argument, false otherwise).

- Write two versions of `sorted` , using recursion and tail-recursion, respectively
- Write an invocation of your function `sorted` to check that a given list of integers is sorted.