TECNOLOGIAS
E ARQUITETURA

List, Generic Methods, Recursion and Pattern Matching

Partially based on Martin Odersky slides

Functional Programming

Scala

```
def factorial(n: Int): Int = n match {
    case 0 => 1
    case _ => n * factorial(n-1)
}
```

Java

```
int factorial(int n) {

 int i=1, r=1;
 while(i <= n) {
    r = r * i;
    i = i + 1;
 }
 return r
}
```

Both sides of operator **=>** have the same value

Very different from the operator **=** in imperative programming

For instance, the instruction `i = i + 1` represents an attribution (the previous value of i is destroyed – the new value is the previous plus one).
The value of `i` is redefined.

In **Pure Functional Programming** it is not possible to redefine, so we reason about equations

Scala Substitution Model

Scala model of **expression evaluation** is based on the principle of substitution model

- Variable names are replaced by the values they are bound to
- Idea: all evaluation reduce an expression to a value
  - Values includes constants, tuple, constructors, and functions

In Scala program expressions are evaluated in the same way we would evaluate a mathematical expression, e.g. (2∗2)+(3∗4)

- first evaluating 2∗2
- then 3∗4
- finally, 4+12

Expression Evaluation

- Expression evaluation works by rewriting the original expression
- Rewriting works by performing simple steps called **reductions**

```
scala> def x = 2
scala> def y = 5
scala> (2 * x) + (4 * y)
```

$$\to (2 * 2) + (4 * y)$$

$$\to 4 + (4 * 5)$$

$$\to 4 + 20$$

$$\to 24$$

1. Take the operator with highest precedence.
2. Evaluate its operands from left to right.
3. Apply the operator to operand values.

Function (with parameters) Evaluation

1. Evaluates **all** the function **arguments** from left-to-right.
2. Replace the function application by the functions right-hand side, and, at the same time
3. Replace all the formal parameters of the function by the actual arguments.

```scala
scala> def square(x: Double) = x * x
scala> square(3 + 3)
```

$$\rightarrow square(6)$$

$$\rightarrow 6 * 6$$

$$\rightarrow 36$$

Call-by-value and *Call-by-name*

```
scala> def sumofSquares(x: Int, y:Int) = square(x) + square(y)
def sumofSquares(x: Int, y: Int): Double
```

1. *call-by-value*: *Call-by-value* evaluates every function argument only once thus it avoids the repeated evaluation of arguments. Example: $\rightarrow sumofSquares(2, 2+3)$

   $\rightarrow square(2) + square(5)$

   $\rightarrow 2*2+5*5$

   $\rightarrow 4+25$

   $\rightarrow 29$

2. *call-by-name*: *Call-by-name* avoids evaluation of parameters if it is not used in the function body. Example:

   $\rightarrow square(2) + square(2+3)$

   $\rightarrow square(2) + square(2+3)$

   $\rightarrow 2*2+square(2+3)$

   $\rightarrow 4+(2+3)*(2+3)$

   $\rightarrow 4+5*(2+3)$

   $\rightarrow 4+5*5$

   $\rightarrow 4+25$

   $\rightarrow 29$

Call-by-value and *Call-by-name*

Scala uses ***call-by-value* as a default**

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

Then the evaluation of function `test(1, loop)` is:

1. *Call-by-name* evaluation reduces to 1.

   $\rightarrow 1$

2. *Call-by-value* evaluation leads to infinite loop.

   $\rightarrow test(1, loop)$

   $\rightarrow test(1, loop)$

   $\rightarrow \ldots$

We can force it to use *call-by-name* by preceding the parameter types by ⇒. Example:

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

Generic Methods

```
def listOfDuplicates[A](x: A, length: Int): List[A] = {
  if (length < 1) Nil //List()
  else x :: listOfDuplicates(x, length - 1)
}

println(listOfDuplicates[Int](3, 4)) // List(3, 3, 3, 3)
```

It is also not compulsory to specify type parameters when generic methods are called.

```
println(listOfDuplicates("La", 4)) // List(La, La, La, La)
```

Type Inference

The Scala compiler can **often** infer the type of an expression, so you don't have to declare it explicitly.

```scala
val businessName = "Jazz Coffee“

def squareOf(x: Int) = x * x
```

Some Collections in more detail

- **Lists**
- **Range**

Collections - List

List is a fundamental data structure in functional programming

Examples

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

Lists are immutable – the elements of a list cannot be changed

You have still operations that **simulate modifications**, but will return a new collection and leave the **old collection unchanged**

List - Example

Example

```
scala> val nums = List(1, 2, 3)
scala> nums.updated(2,9) //nums updated(2,9) √
scala> ??
scala> nums
scala> ??
scala> nums = List(0)  //??
scala> var nums = List(1, 2, 3) //avoid in pure FP
scala> nums = List(0)
scala> nums
scala> ??
```

List vs Array - example

```
scala> val x1 = Array(1,2,3)
val x1: Array[Int] = Array(1, 2, 3)

scala> val x1 = Array(1,2,3) updated(2,9)

scala> x1
val res55: Array[Int] = Array(1, 2, 9)
```

But Array, being **mutable,** should be avoided in Pure FP

List - construction

All lists are constructed from:

- the empty list `Nil`, and
- the construction operation `::` (pronounced *cons*):
  `x :: xs` gives a new list with the first element `x`, followed by the elements of `xs`.

For example:

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

new Cons(x, xs)

x :: xs

Operations on Lists

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

`head`	the first element of the list
`tail`	the list composed of all the elements except the first.
`isEmpty`	‘true‘ if the list is empty, ‘false‘ otherwise.

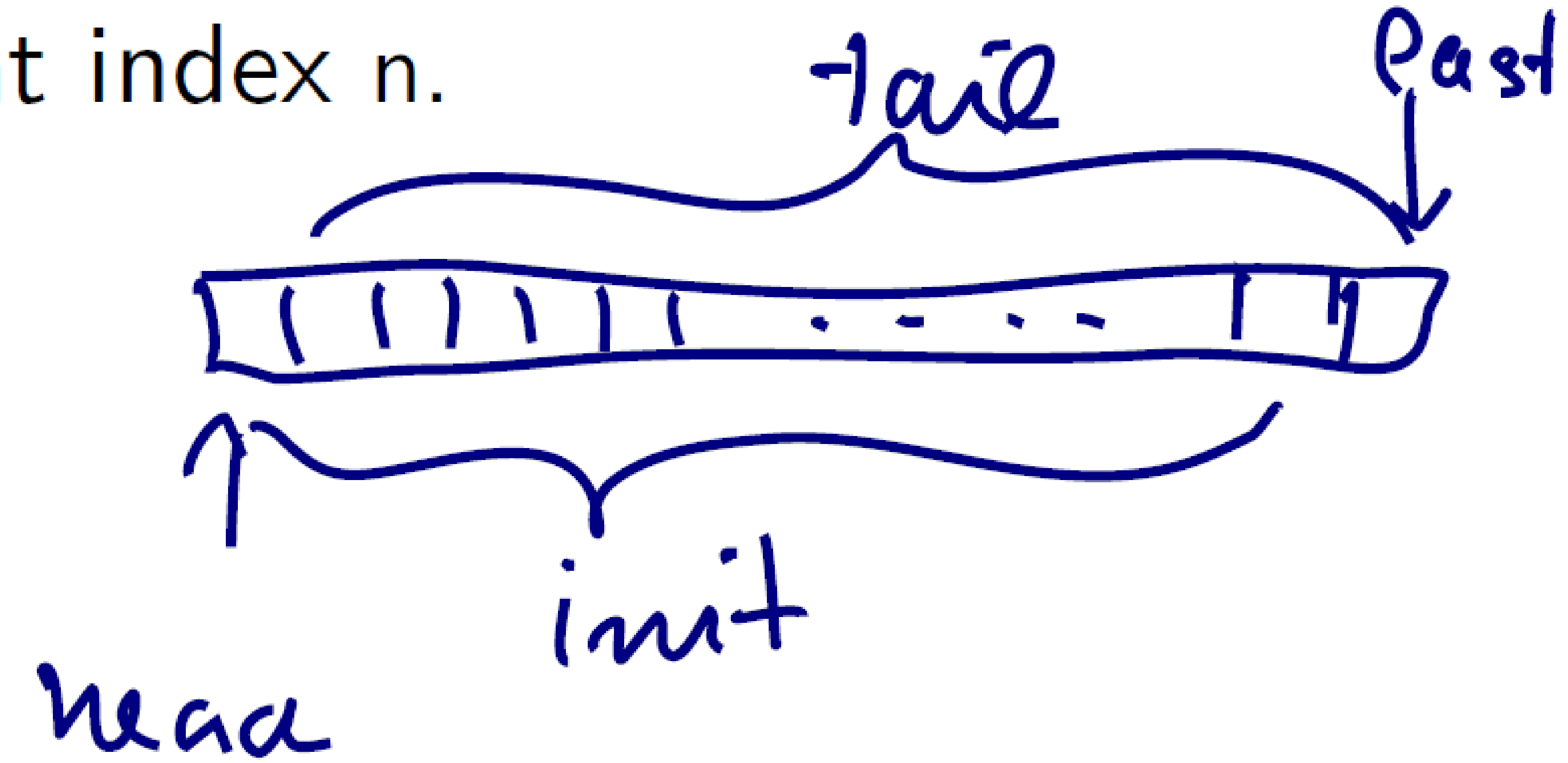
These operations are defined as methods of objects of type list. For example:

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

More functions on Lists

Sublists and element access:

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



More functions on Lists

Adding element at the end:

```
List(2,3) :+ 4
(1 :: 2 :: Nil) :+ 4
```

Creating new lists:

`xs ++ ys` `xs ::: ys`	The list consisting of all elements of `xs` followed by all elements of `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`.

Finding elements:

`xs indexOf x`	The index of the first element in `xs` equal to `x`, or `-1` if `x` does not appear in `xs`.
`xs contains x`	same as `xs indexOf x >= 0`

More functions on Lists

Transposing a List of Lists

```
scala> val x = List( List(1, 2, 3), List(4, 5, 6))
scala> x.transpose
val res11 = List[List[Int]] = List(List(1, 4), List(2, 5), List(3, 6))
```

Collections - Range

```
scala> 1 to 10
res0: scala.collection.immutable.Range.Inclusive = Range(1, 2, 3, 4, 5,
6, 7, 8, 9, 10)

scala> 1 until 10
res1: scala.collection.immutable.Range = Range(1, 2, 3, 4, 5, 6, 7, 8,
9)

scala> 1 to 10 by 2
res2: scala.collection.immutable.Range = Range(1, 3, 5, 7, 9)

scala> 'a' to 'c’
res3: collection.immutable.NumericRange.Inclusive[Char] =
NumericRange(a, b, c)
```

Range

```
scala> val x = (1 to 10).toList
x: List[Int] = List(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val x = (1 to 10).toArray
x: Array[Int] = Array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

scala> val x = (1 to 10).toSet
x: scala.collection.immutable.Set[Int]=Set(5, 10, 1, 6, 9, 2, 7, 3, 8, 4)
```

Recursion (or Recursivity)

How to obtain the length of a list?

Two cases:
- list empty: the length is zero
- list not empty: the length is one plus the length of the tail

```
def length[E](l: List[E]):Int = {
  if(l.isEmpty) 0
  else 1 + length(l.tail)
}
```

The method is recursive as it **calls itself**

Recursion

The method **ends** because the recursive calls are made on increasingly shorter lists until it is empty.

```
length(List(1,2,3)) =
 length(1::List(2,3)) =>
 1 + length(List(2,3)) =>
 1 + (1 + length(List(3)) =>
 1 + (1 + (1 + length(List()))) =>
 1 + (1 + (1 + 0)) =>
 3
```

Recursion - notes

Recursive methods need an **explicit return type** in Scala. For non-recursive methods, the return type is optional.

For recursive methods, the compiler is not able to infer a result type. Here is a method that **will fail** to compile:

```
def fac(n: Int) = if (n == 0) 1 else n * fac(n - 1)
```

```
How to solve it?
```

In FP recursion is a way of performing loops

Loops (e.g., for) exist in Scala but should not be used in Pure FP

Pattern Matching

A function can have multiple patterns

Almost like overloading methods in Java. More powerful version of `switch` (Java). It's expressed using the keyword `match`

wildcard pattern

```
def guess(x : Int): String = x match {
 case 42 => "correct!"
 case y  => "wrong guess!" // case _  => "wrong guess!"
}
```

Each pattern has the same type declaration

Pattern Matching - rules

- Patterns are matched in **order, top-down**
- Only the **first** matched pattern is evaluated
- The patterns must exhaust the **entire domain**
  - A `MatchError` exception is thrown if no pattern is matched

What's wrong with this code?

```
def fib(n:Int):Int = n match {
  case _ => fib(n-1) + fib(n-2)
  case 0 => 1
  case 1 => 1
}
```

What's wrong with this code?

```
def fib(n:Int):Int = n match {
 case _ => fib(n-1) + fib(n-2)
 case 0 => 1
 case 1 => 1
}
```

The base case is never hit.
The first pattern eats up everything!
∞ loop

More Pattern Matching

You can even match lists using the Construction operation **::**

```
def head[E](x: List[E]):E = x match{
   case Nil => throw new Error("head of empty list")
   case (firstItem :: everythingElse) => firstItem
}

def tail[E](lst: List[E]):List[E] = lst match{
   case Nil => throw new Error("tail of empty list")
   case (x::xs) => xs
}
```

List Patterns - examples

```
1 :: 2 :: xs
x :: Nil
List(x)
List()
List(2 :: xs)
List(a :: b :: Nil)
List(a :: b :: xs)
```

Pattern Guards

boolean expressions used to make *cases* more specific

Just add *`if<boolean expression>`* after the pattern

```
...
num match {
   case x if(x == 1) => println("one, a lonely number")
   case x if(x == 2 || x == 3) => println(x)
   case _ => println("some other value")
}
```

Equals, *==* and reference equality (rarely used) *Eq*

```
scala> def met() = {
         val x = List(1)
         val y = List(1)
         x equals y
       }
```

```
scala> def met() = {
         val x = List(1)
         val y = List(1)
         x == y
       }
```

```
scala> def met() = {
         val x = List(1)
         val y = List(1)
         x eq y
       }
```

== routes to equals
Different from Java where == is usually used with primitive types

Exercice

Write a function to detect if a list is a palindrome (pt: capicua)
- with pattern matching
- without pattern matching

```
def isPal[E](x: List[E]):Boolean = …
```

Sorting Lists

Suppose we want to sort a list of numbers in ascending order:

- One way to sort the list `List(7, 3, 9, 2)` is to sort the tail `List(3, 9, 2)` to obtain `List(2, 3, 9)`.
- The next step is to insert the head `7` in the right place to obtain the result `List(2, 3, 7, 9)`.

This idea describes *Insertion Sort* :

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

Exercise

Complete the definition insertion sort by filling in the ???s in the definition below:

```
def insert(x: Int, xs: List[Int]): List[Int] = xs match {
  case List() => ???
  case y :: ys => ???
}
```

Exercises: last, concatenation and reverse

```
def last[T](xs: List[T]): T = xs match {
  case List() => throw new Error(”last of empty list”)
  case List(x) =>
  case y :: ys =>
}
def concat[T](xs: List[T], ys: List[T]) = xs match {
  case List() =>
  case z :: zs =>
}
def reverse[T](xs: List[T]): List[T] = xs match {
  case List() =>
  case y :: ys =>
}
```

"In order to understand recursion, one must first understand recursion."

— Anonymous