Introduction to Functional Programming with Scala

Angelo Gargantini

INFO 3A AA 2015/16
credits: Pramode C.E
https://class.coursera.org/progfun-00

April 4, 2016

Workshop Plan

Here is what we will do:

- Learn a bit of functional programming in Scala
- Learn some important concepts like (NOT ALL): closures, higher order functions, purity, lazy vs strict evaluation, currying, tail calls/TCO, immutability, persistent data structures, type inference etc!

Workshop material (slide/code samples) sul sito.

Function Definition

```
def add(a:Int, b:Int):Int = a + b

val m:Int = add(1, 2)

println(m)
```

Note the use of the type declaration "Int". Scala is a "statically typed" language. We define "add" to be a function which accepts two parameters of type Int and returns a value of type Int. Similarly, "m" is defined as a variable of type Int.

Function Definition

```
def fun(a: Int):Int = {
  a + 1
  a - 2
  a * 3
}

val p:Int = fun(10)
println(p)
```

Note!

There is no explicit "return" statement! The value of the last expression in the body is automatically returned.

Type Inference

```
def add(a:Int, b:Int) = a + b

val m = add(1, 2)

println(m)
```

We have NOT specified the return type of the function or the type of the variable "m". Scala "infers" that!

Type Inference

```
def add(a, b) = a + b

val m = add(1, 2)

println(m)
```

This does not work! Scala does NOT infer type of function parameters, unlike languages like Haskell/ML. Scala is said to do local, "flow-based" type inference while Haskell/ML do Hindley-Milner type inference

References

1. http://en.wikipedia.org/wiki/Hindley-Milner
2. http://www.scala-lang.org/node/4654

Expression Oriented Programming

```
val i = 3
val p = if (i > 0) -1 else -2
val q = if (true) "hello" else "world"

println(p)
println(q)
```

Unlike languages like C/Java, almost everything in Scala is an "expression", ie, something which returns a value! Rather than programming with "statements", we program with "expressions"

Expression Oriented Programming

```
def errorMsg(errorCode: Int) = errorCode match {
  case 1 => "File not found"
  case 2 => "Permission denied"
  case 3 => "Invalid operation"
}

println(errorMsg(2))
```

Case automatically "returns" the value of the expression corresponding to the matching pattern.

Evaluation

Applications of parametrized functions are evaluated in a similar way as operators. Expressions are evaluaed before passing their value to functions (values are passed to functions)

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

square(2)

square(2+2)

square(square(2))
```

Evaluation of function application

Given a function application $f(e_1, \ldots, e_n)$

1. Evaluate all function arguments $(e_1, \ldots, e_n)$ from left to right. Let $v_1, \ldots, v_n$ the corresponding values.
2. Replace the function application by the function's right hand side (function body), and, at the same time
3. Replace (substitute) the formal paramters of the function by the actual arguments $v_1, \ldots, v_n$

```
def sumOfSquare(x:Double, y: Double) = square(x) +
    square(y)
```

```
sumOfSquare(3,2+2)
sumOfSquare(3,4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
....
```

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*
The basic idea is to reduce an expression to a value
It can proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

Evaluation of function application

This scheme of expression evaluation is called the *substitution model*
The basic idea is to reduce an expression to a value
It can proved that this model can represent any algorithm (except side effect).

Termination

Does every expression reduce to a value (in a finite number of steps)? NO

```
def loop: Int = loop

loop
```

Alternative evaluation

The interpreter reduces function arguments to values before rewriting the function application.
One could alternatively apply the function to unreduced arguments.
For instance:

```
sumOfSquare(3,2+2)
square(3) + square(2+2)
3*3 + (2+2) * (2+2)
....
```

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4) CBN faster

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4) CBN faster
test(3+4,2*4)

Pass by-name

The first strategy is call-by-value, the second is *call-by-name* (they are equivalent if they terminate - but CBN may terminate)

- call-by-value: evaluates every function argument only once
- call-by-name: a function is evaluated only if necessary

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

which call is fastest (fewest num. operations)?

test(2,3) same number of steps
test(3+4,8) CBV faster
test(7,2*4) CBN faster
test(3+4,2*4) same number of steps

Call by name functions

Scala normally uses call-by-value
But if the type of a function parameter with $=>$ it uses call-by-name
Example

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

Using pen and paper, trace the evaluation of the following function calls for the function constOne:
constOne(1+2,loop)
constOne(loop,1+2)

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Recursion

- Recursion means a function can call itself repeatedly.
- Recursion plays a big role in pure functional programming and Scala supports recursion functions very well.

Consider the Eulid's algorithm

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

evaluate gcd(14,21) ...
Consider the factorial algorithm

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

evaluate factorial(4) ...

Recursion

```
// sum n + (n-1) + (n-2) + ... + 0
def sum(n: Int): Int =
  if (n == 0) 0 else n + sum(n - 1)

val m = sum(10)
println(m)
```

Try calling the function "sum" with a large number (say 10000) as parameter! You get a stack overflow!

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion.
The function's stack frame can be reused (Tail Call Optimization).
Rewrite the function as tail recursion.

Tail Calls and TCO

If a function calls itself as its last action is called tail recursion.
The function's stack frame can be reused (Tail Call Optimization).
Rewrite the function as tail recursion.

```
def factorial(n: Int): Int = {
  def loop(acc: Int, n: Int)=
     if ( n == 0) acc
     else loop(n*acc, n-1)
  loop(0,n)
}
```

Tail Calls and TCO

```
def sum(n: Int, acc: Int):Int =
  if(n == 0) acc else sum(n - 1, acc + n)

val r = sum(10000, 0)

println(r)
```

This is a "tail-recursive" version of the previous function - the Scala compiler converts the tail call to a loop, thereby avoiding stack overflow!

Proviamo con la segnatura semplificata e un loop interno.

Tail Calls and TCO

```
(sum 4)
(4 + sum 3)
(4 + (3 + sum 2))
(4 + (3 + (2 + sum 1)))
(4 + (3 + (2 + (1 + sum 0))))
(4 + (3 + (2 + (1 + 0))))
(4 + (3 + (2 + 0)))
(4 + (3 + 2))
(4 + 5)
(9)
```

```
(sum 4 0)
(sum 3 4)
(sum 2 7)
(sum 1 8)
(sum 0 9)
(9)
```

Higher-Order Functions

- Functional languages treat functions as *first-class values*
- This means that, like any other value, a function can be passed as a parameter and returned as a result
- Functions that take values and variable are called *first order functions*
- Functions that take other functions as parameters or return functions are called *higher order functions*

Summation once again!

take the sum of the integers from a and b:

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

Summation once again!

take the sum of the integers from a and b:

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

If you want to sum the squares or cubes from a and b:

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

def cube(x: Int) = x * x * x
def sumCubes(a: Int, b: Int): Int =
  if (a > b) 0 else cube(a) + sumCubes(a + 1, b)
```

Summation

Exercise

Define the sum of factorial from a and b

Summation

Exercise

Define the sum of factorial from a and b

Idea

Define a sum generic with the respect to the operation applied to each number?

```
def operation(x:Int) = ...

def sumOperation(a: Int, b: Int) =
  if (a > b) 0 else operation(a) + sumOperation(a+1,b)
```

Higher order functions

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

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type `Int => Int` represents a function which accepts an Int and returns an Int.

Higher order functions

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

"sum" is now a "higher order" function! It's first parameter is a function which maps an Int to an Int

The type "Int" represents a simple Integer value. The type `Int => Int` represents a function which accepts an Int and returns an Int.

```
def identity(x: Int) = x
def sqr(x: Int) = x * x
def cube(x: Int) = x * x * x
def fact(x:Int) = ...
println(sum(identity, 1, 10))
println(sum(sqr, 1, 10))
println(sum(cube, 1, 10))
```

Anonymous functions

Passing functions as parameters leads to the creation of many functions. Sometime is tedious. It can be avoided.
Like:

```
def name = "Angelo"; println(name)
```

can be written as

```
println("Angelo)
```

we want to define functions without an explicit name:
anonymous functions

Anonymous functions

Can be written as:
(*Parameters*) => *Body*

We can create "anonymous" functions on-the-fly! `x => x*x` is a function which takes an "x" and returns x*x

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

The parameter type can be omitted if the compiler can infer it:

```
x => x *x
```

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

Higher order functions and recursive calls

Rewrite sum with the tail recursion?

Higher order functions and recursive calls

Rewrite sum with the tail recursion?

```
def sum(f: Int => Int, a: Int, b: Int): Int = {
    def loop(a: Int, acc: Int): Int = {
        if (a > b) acc
        else loop(a + 1, acc + f(a))
    }
    loop(a, 0)
}
```

Currying

Here is the definition from Wikipedia:

> In mathematics and computer science, currying is the technique of transforming a function that takes multiple arguments (or a tuple of arguments) in such a way that it can be called as a chain of functions, each with a single argument. It was originated by Moses Schonfinkel and later re-discovered by Haskell Curry.

Let's try to do this in Scala!

Currying - returning functions

```
def sumInts(a:Int, b: Int) = sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) = sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) = sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

Currying - returning functions

```
def sumInts(a:Int, b: Int) = sum(x=>x, 1, 10)
def sumCubes(a:Int, b: Int) = sum(x=>x*x, 1, 10)
def sumFactorial(a:Int, b: Int) = sum(fact, 1, 10)
```

a and b are passed to sum unchanged. Can we rewrite sum and avoid the use of a and b?

sum can *return* a function that takes two Ints and return an Int

```
def sum(f: Int => Int): (Int,Int) => Int = { ...}
```

sum takes a function `f` and return a function `(Int,Int) => Int`

Currying - returning functions

```
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
}
```

Currying - stepwise application

The basic sum functions can be defined without parameters:

```
def sumInts = sum(x=>x)
def sumCubes = sum(x=>x*x)
def sumFactorial = sum(fact)
```

sumInts(3,4) ...
or we could write

```
sum(x=>x)(3,4)
```

Multiple parameter list

```
def sum(f: Int => Int, a: Int, b: Int): Int =
```

Can be rewritten as:

```
def sum(f: Int => Int) : (Int, Int) => Int =
```

or equivalently, by using multiple parameter lists:

```
def sum(f: Int => Int)(a: Int, b: Int): Int =
```

the advantage wrt the first is that se can pass only one argument like sum(cube) the advantage wrt the second is that we can use a and b directly in the body

Currying - two argument functions

```
def addA(x: Int, y: Int): Int =
  x + y

def addB(x: Int):Int=>Int =
  y => x + y

val a = addA(10, 20)

val b = addB(10)(20)

println(a)
println(b)
```

Currying - three argument functions

```
def addA(x: Int, y: Int, z: Int) = x + y + z

def addB(x: Int): Int => (Int => Int) =
  y => (z => x + y + z)

val a = addA(1, 2, 3)

val b = addB(1)(2)(3)

println(a)
println(b)
```

It is now easy to see how the idea can be generalized to N argument functions!

Exercise

1. write a function that calculates the product of the values for the points on a gievn interval
2. write a function that calculates the product of the values of a function f for the points on a given interval
3. write the factorial in terms of product
4. can we write a more general function which generalizes both sum and product

Exercise

1. write a function that calculates the product of the values for the points on a gievn interval
2. write a function that calculates the product of the values of a function f for the points on a given interval
3. write the factorial in terms of product
4. can we write a more general function which generalizes both sum and product

END 1

Methods on collections: Map/Filter/Reduce

```
val a = List(1,2,3,4,5,6,7)

val b = a.map(x => x * x)
val c = a.filter(x => x < 5)
val d = a.reduce((x, y)=>x+y)

println(b)
println(c)
println(d)
```

Map applies a function on all elements of a sequence. Filter selects a set of values from a sequence based on the boolean value returned by a function passed as its parameter - both functions return a new sequence. Reduce combines the elements of a sequence into a single element.

More methods on collections

```
def even(x: Int) = (x % 2) == 0

val a = List(1,2,3,4,5,6,7)
val b = List(2, 4, 6, 5, 10, 11, 13, 12)

// are all members even?
println(a.forall(even))

// is there an even element in the sequence?
println(a.exists(even))

//take while the element is even -
//stop at the first odd element
println(b.takeWhile(even))

//partition into two sublists: even and odd
println(a.partition(even))
```

Block structure / Scope

```
def fun(x: Int) = {
  val y = 1

  val r = {
    val y = 2
    x + y
  }
  println(r)
  println(x + y)
}

fun(10)
```

The "y" in the inner scope shadows the "y" in the outer scope

Nested functions / functions returning functions

```
// fun returns a function of type Int => Int

def fun():Int => Int = {
  def sqr(x: Int):Int = x * x

  sqr
}

val f = fun()
println(f(10))
```

`def fun():Int=>Int` says "fun is a function which does not take any argument and returns a function which maps an Int to an Int. Note that it possible to have "nested" function definitions.

Lexical Closure

```
def fun1():Int => Int = {
  val y = 1
  def add(x: Int) = x + y

  add
}

def fun2() = {
  val y = 2
  val f = fun1()

  // what does it print? 11 or 12
  println(f(10))
}

fun2()
```

Lexical Closure

- The function "fun1" returns a "closure".
- A "closure" is a function which carries with it references to the environment in which it was defined.
- When we call f(10), the "add" function gets executed with the environment it had when it was defined - in this environment, the value of "y" is 1.

Lexical Closure with anonymous functions

```
def fun1(y: Int):Int=>Int =
  x => x + y

def fun2() = {
  val f = fun1(10)
  println(f(2))
}

fun2()
```

- "y" is now a parameter to fun1
- "fun1" returns an anonymous function - there is absolutely no difference between returning a "named" function and returning an anonymous function.

Simple closure examples

```
def sqr(x: Int) =  x*x
def cube(x: Int) =  x*x*x

def compose(f: Int=>Int, g: Int=>Int): Int=>Int =
  x => f(g(x))

val f = compose(sqr, cube)
println(f(2))

val a = List(1,2,3,4)

println(a.map(f))

println(a.map(cube).map(sqr))
```

Simple closure examples

```
def removeLowScores(a: List[Int],
                    threshold: Int): List[Int] =
  a.filter(score => score >= threshold)

val a = List(95, 87, 20, 45, 35, 66, 10, 15)

println(removeLowScores(a, 30))
```

- The anonymous function "`score => score >= threshold`" is the closure here.
- How do you know that it is a closure? Its body uses a variable "threshold" which is not in its local environment (the local environment, in this case, is the parameter list consisting of a single parameter "score")

Some List operations

```
val a = List(1,2,3)
val b = Nil
val c = List()
val d = 0::a
val e = 0::b

println(b)
println(c)
println(d) // List(0,1,2,3)
println(e) // List(0)
```

- Nil and List() are both "empty" lists
- a::b returns a new list with "a" as the first item (the "head") and remaining part b (called the "tail")

Non-strict evaluation

```
def myIf(cond: Boolean, thenPart: Int, elsePart: Int) =
  if (cond) thenPart else elsePart


println(myIf((1 < 2), 10, 20))
```

We are trying to write a function which behaves similar to the built-in "if" control structure in Scala ... does it really work properly? Let's try another example!

Non-strict evaluation

```
def fun1() = {
  println("fun1")
  10
}
def fun2() = {
  println("fun2")
  20
}
def myIf(cond: Boolean, thenPart: Int, elsePart: Int) =
  if (cond) thenPart else elsePart

println(myIf((1 < 2), fun1(), fun2()))
```

Non-strict evaluation

- The behaviour of "if" is "non-strict": In the expression "if (cond) e1 else e2", if "cond" is true e2 is NOT EVALUATED. Also, if "cond" is false, e1 is NOT EVALUATED.
- By default, the behaviour of function calls in Scala is "strict": In the expression "fun(e1, e2, ..., en), ALL the expressions e1, e2 ... en are evaluated before the function is called.
- There is a way by which we can make the evaluation of function parameters non-strict. If we define a functions as `"def fun(e1: => Int)"`, the expression passed as a parameter to "fun" is evaluated ONLY when its value is needed in the body of the function. This is the "call-by-name" method of parameter passing, which is a "non-strict" strategy.

Non-strict evaluation

```
def fun1() = {
  println("fun1")
  10
}
def fun2() = {
  println("fun2")
  20
}

def myIf(cond: Boolean, thenPart: => Int,
         elsePart: => Int) =
  if (cond) thenPart else elsePart

println(myIf((1 < 2), fun1(), fun2()))
```

Non-strict evaluation

```
def hello() = {
  println("hello")
  10
}

def fun(x: => Int) = {
  x + x
}

val t = fun(hello())
println(t)
```

How many times is the message "hello" printed? Is there some way to prevent unnecessary repeated evaluations?

Lazy val's

```
def hello() = {
  println("hello")
  10
}

val a = hello()
```

The program prints "hello" once, as expected. The value of the val "a" will be 10.

Lazy val's

```
def hello() = {
  println("hello")
  10
}

lazy val a = hello()
```

Strange, the program does NOT print "hello"! Why? The expression which assigns a value to a "lazy" val is executed only when that lazy val is used somewhere in the code!

Lazy val's

```
def hello() = {
  println("hello")
  10
}

lazy val a = hello()

println(a + a)
```

Unlike a "call-by-name" parameter, a lazy val is evaluated only once and the value is stored! This is called "lazy" or "call by need" evaluation.

Referential Transparency

If an expression can be replaced by its value without changing the behaviour of the program, it is said to be referentially transparent

- All occurrences of the expression 1+(2*3) can be replaced by 7 without changing the behaviour of the program.
- Say the variable x (in a C program) has initial value 5. It is NOT possible to replace all occurrences of the statement ($x = x + 1$) with the value 6.

Pure Functions

```
var balance = 1000

def withdraw(amount: Int) = {
  balance = balance - amount
  balance
}

println(withdraw(100))
println(withdraw(100))
```

In what way is our "withdraw" function different from a function like "sin"?

Pure Functions

- A pure function always computes the same value given the same parameters; for example, sin(0) is always 0. It is "Referentially Transparent".
- Evaluation of a pure function does not cause any observable "side effects" or output - like mutation of global variables or output to I/O devices.

What is Functional Programming?

A style of programming which emphasizes composing your program out of PURE functions and immutable data.

Theoretical foundation based on Alonzo Church's Lambda Calculus

In order for this style to be effective in the construction of real world programs, we make use of most of the ideas seen so far (higher order functions, lexical closures, currying, immutable and persistent datastructures, lazy evaluation etc)

What is Functional Programming?

Questions to ask:

- Is this the "silver bullet"?
- How practical is a program composed completely out of "pure" functions?
- What are the benefits of writing in a functional style?

Is FP the silver bullet?

- Of course, there are NO silver bullets! (`http://en.wikipedia.org/wiki/No_Silver_Bullet`)
- Writing software is tough - no one methodology or technique is going to solve all your problems

How practical is a program composed completely out of pure functions?

- Very impractical - unless your aim is to fight the chill by heating up the CPU (which, by the way, is a "side effect")
- The idea is to write your program in such a way that it has a purely functional core, surrounded by a few "impure" functions at the outer layers

Functional core + "impure" outer layers

This example is simple and contrived, but it serves to illustrate the idea. It is taken from the amazing book "Functional Programming in Scala" by Paul Chiusano and Runar Bjarnason.

```
case class Player(name: String, score: Int)

def declareWinner(p: Player) =
  println(p.name + " is the winner!! ")

def winner(p1: Player, p2: Player) =
  if(p1.score > p2.score) declareWinner(p1)
  else declareWinner(p2)

winner(Player("Ram", 10), Player("John", 20))
```

Note that "winner" is an impure function. We will now refactor it a little!

Functional core + "impure" outer layers

```
case class Player(name: String, score: Int)

def declareWinner(p: Player) =
  println(p.name + " is the winner!! ")

def maxScore(p1: Player, p2: Player) =
  if (p1.score > p2.score) p1 else p2

def winner(p1: Player, p2: Player) =
  declareWinner(maxScore(p1, p2))

winner(Player("Ram", 10), Player("John", 20))
```

Now we have separated the computation from the display logic; "maxScore" is a pure function and "winner" is the impure function at the "outer" layer!

Benefits of functional style - easy reuse, easy testing

What if we wish to find out the winner among a set of N players? Easy!

```
val players = List(Player("Ram", 10),
                   Player("John", 15),
                   Player("Hari", 20),
                   Player("Krishna", 17))

println(players.reduceLeft(maxScore))
```

FP as "good software engineering"

- Pure functions are easy to re-use as they have no "context" other than the function parameters. (Think about re-using the "winner" function in our first version to compute the winner among N players).
- Pure functions are also easy to test. (Think about writing an automated test for the "winner" function in the first version).
- Think of FP as "Good Software Engineering"!

Pure functions and the benefit of "local reasoning"

- If your function modifies an object which is accessible from many other functions, the effect of calling the function is much more complex to analyse because you now have to analyse how all these other functions get affected by the mutation.
- Similarly, if the value computed by your function depends on the value of an object which can be modified by many other functions, it no longer becomes possible to reason about the working of the function by only looking at the way the function's parameters are manipulated.
- The evaluation of pure functions can be done by a very simple process of "substitution".

Why mutability is tricky - an example

```
class ProtectedResource {
  private Resource theResource = ...;
  private String [] allowedUsers = ...;
  public String[] getAllowedUsers() {
    return allowedUsers;
  }
  public String currentUser() { ... }
  public void useTheResource() {
    for(int i = 0; i < allowedUsers.length; i++) {
      if(currentUser().equals(allowedUsers[i]) {
        ... // access allowed; use it!
      }
    }
    throw new IllegalAccessException();
  }
}
```

Why mutability is tricky - an example

This Java example is taken from Prof.Dan Grossman's (University of Washington) excellent coursera.org class on "Programming Languages". Can you identify the problem with the code?

Why mutability is tricky - an example

What if client code does this?

```
p.getAllowedUsers[0] = p.currentUser()
p.useTheResource()
```

The user can happily use the resource, even if he does not belong to the group of "allowed" users! The fix is to return a copy of "allowedUsers" in the function "getAllowedUsers".

What if we had used an immutable Scala list for representing "allowedUsers"? This problem would never had occurred because an attempt to modify "allowedUsers" simply returns a new object without in any way altering the original!

How it gets even more tricky in the context of concurrency

- Multi core CPU's are becoming commonplace
- We need concurrency in our code to make effective use of the many cores
- This throws up a whole bunch of complex problems
- A function can no longer assume that nobody else is watching when it is happily mutating some data

How it gets even more tricky in the context of concurrency

```
case class Date(var year: Int, var month: String,
                var date: Int, var weekDay: String)

val d = Date(2013, "February", 23, "Saturday")
def changeDate(year: Int, month:String,
               date: Int, weekDay: String) = {
  d.year = year
  d.month = month
  d.date = date
  d.weekDay = weekDay
}
def showDate() = println(d)
changeDate(2013, "February", 24, "Sunday")
showDate()
```

How it gets even more tricky in the context of concurrency

What happens if the functions changeDate() and showDate() run as two independent threads on two CPU cores? Will showDate() always see a consistent date value?

The traditional approach to maintaining correctness in the context of multithreading is to use locks - but people who do it in practice will tell you it is extremely tricky business.

It is claimed that one of the reasons for the resurgence of FP is the emergence of multi-core processors and concurrency - it seems like FP's emphasis on pure functions and immutability is a good match for concurrent programming.

Moving ahead ...

- Join Prof.Grossman's programming languages class on Coursera: `https://www.coursera.org/course/proglang` and learn more about functional programming using SML and Racket
- Join Prof.Odersky's Functional Programming with Scala course on Coursera for an introduction to both Scala and FP: `https://www.coursera.org/course/progfun`
- Watch the classic "SICP" lectures by Abelson and Sussman: `http://groups.csail.mit.edu/mac/classes/6.001/abelson-sussman-lectures/`
- Learn from "HTDP" if you want something simpler: `http://htdp.org/`

Books

- "Programming in Scala, 2nd edition" by Martin Odersky, Bill Venners, Lex Spoon. Perhaps the best introductory book on the language.
- "Functional Programming in Scala" by Paul Chiusano and Runar Bjarnson - this will soon become a classic!
- "Scala in Depth" by Joshua Suereth. This is an advanced book on Scala
- "Learn You a Haskell for Great Good" - `http://learnyouahaskell.com/`. Scala programmers can definitely benefit from an understanding of Haskell - this is an amazing book which will get you started with Haskell.
- Many others: check out `http://blog.typesafe.com/week-of-scala-with-manning-publications`

Other resources

- "Learning functional programming without growing a neckbeard" - `http://marakana.com/s/post/1354/learning_functional_programming_scala_video`
- "Scala Days 2012" Videos - `http://skillsmatter.com/event/scala/scala-days-2012`
- "Out of the tar pit" - paper by Ben Mosely and Peter Marks (google it)
- "Persistent Data Structures and Managed References" - talk by Rich Hickey (author of the Clojure programming language). `http://www.infoq.com/presentations/Value-Identity-State-Rich-Hickey`
- "Can programming be liberated from the von Neumann style" - by John Backus. `http://www.thocp.net/biographies/papers/backus_turingaward_lecture.pdf`