Ejercicios en Scala

J.A. Medina

Ciencias de la Computación

Universidad de Alcalá

Funciones de listas

- Inicializacion de vals con listas

```
val fruit = List("apples", "oranges", "pears")
val nums = List(1, 2, 3, 4)
val diag3 = List(List(1, 0, 0), List(0, 1, 0), List(0, 0, 1))
val empty = List()
```

- Operador Cons

```
val dosTres = List(2, 3)
val unDosTres = 1 :: dosTres
```

Dada la siguiente lista:

```
val lista = List(1,2,3,4,5,6,7,8,9)
```

- Cuantos elementos cumplen la condición “es par”

```
lista.count((x:Int) => {x % 2 == 0})  //res0: Int = 4
```

- Realiza un método que indique cuantos elementos cumplen la condición “es par” y muestre los que son par

```
def par(x:Int): Boolean = {x % 2 == 0}
lista.count(par _)
lista.filter(par _)                    //List[Int] = List(2, 4, 6, 8)
```

- Realiza un método aplique el x^2 a todos los elementos de la lista

```
def cuadrado(x:Int) = x*x
lista.map(cuadrado _)      // List[Int] = List(1, 4, 9, 16, 25, 36, 49, 64, 81)
```

- Realiza un método convierta en mayúsculas todos los elementos de la lista

```
List("No","es","elegante","escribir","con","mayusculas").map(s => s.toUpperC ase)
 List[String] = List(NO, ES, ELEGANTE, ESCRIBIR, CON, MAYUSCULAS)
```

Cree una lista que llame a tres métodos y devuelva la operación sobre el ultimo elemento de la lista

```
def aplicaLista (lista: List[(Int) => Int], x: Int): Int =
        if (lista.length == 1) lista.head(x)
        else lista.head(aplicaLista(lista.tail,x))


def mas5(x: Int) = x+5
def por8(x: Int) = x*8
def suma3(a: Int, b: Int, c: Int) = a + b + c

val l = List(mas5 _, por8 _, suma3(1, _: Int, 10))

aplicaLista(l, 10)                          => 173
```

Cree una lista que llame a tres métodos y devuelta el valor de la cabeza de la lista

```
def suma3(a: Int, b: Int, c: Int) = a + b + c

def mult3(a: Int, b: Int, c: Int) = a * b * c

val listaFuncs: List[(Int,Int,Int)=>Int] = List(suma3 _, mult3 _, (x:Int,y:Int,z:Int)=>x+y*z)

val f = listaFuncs.head

f(1,2,3)
```

⇒ Int = 6

Realice el sumatorio de 1 a 10, aplicando a cada elemento del sumatorio x+3

Realice el sumatorio de 1 a 10, aplicando a cada elemento del sumatorio x+3

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

sumatorio(1, 10, (x:Int) => {x+3})                    // Res0:   Int=85

def suma3(x:Int) = x+3

sumatorio(1,10,suma3 _)
```

Calcular la lista de elementos entre el 1 y el 5 al cuadrado

Calcular la lista de elementos entre el 1 y el 5 al cuadrado

```
for (x <- List.range(1,6)) yield x*x
```

⇒ List(1, 4, 9, 16, 25)

Expresión devuelva el cuadrado de los números impares divisibles por 3 del 1 al 100

Expresión devuelve el cuadrado de los números impares divisibles por 2 y 3 del 1 al 100

```
for (i <- List.range(1, 101) if (i % 2 != 0 && i % 3 == 0)) yield i*i
```

⇒List[Int] = List(9, 81, 225, 441, 729)

Expresión devuelve una colección de parejas formadas con dos generadores

Expresión devuelve una colección de parejas formadas con dos generadores

```
for(x <- (1 to 3 ); y <- (1 to x)) yield (x,y)
```

⇒ Vector((1,1), (2,1), (2,2), (3,1), (3,2), (3,3))

Calculo del factorial de un número

Calculo del factorial de un numero

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

def fact(n:Int): Int = n match {
        case 0 => 1
        Case n => n *fact (n-1)
        }
Fact(5)
```

Máximo común divisor de dos números

Máximo común divisor de dos números

```
def gcd(x: Long, y:Long): Long =
                    if(y==0) x else gcd(y, x % y)
```

Número de elementos que contiene una lista

Número de elementos que contiene una lista

```
def numElems(l:List[Int]):Int =
                    if (l == Nil) 0 else 1 + numElems(l.tail)


def numElemes(ls:List[Int]): Int = ls match {
        case Nil => 0
        Case l::ls => 1 + numElems(ls)
        }
```

Insertar en una lista ordenada

Insertar en una lista ordenada

```
def insert(x: Int, lista: List[Int]) : List[Int] =
                    if (lista.isEmpty) x :: Nil else
                    if (x < lista.head) x :: lista
                            else lista.head :: insert(x, lista.tail)
```

Ordenación de una lista:

Ordenación de una lista:

```
def sort(lista: List[Int]): List[Int] =
                if (lista.isEmpty) Nil
                else insert(lista.head, sort(lista.tail))
```

Concatenación de dos listas

Concatenación de dos listas

```
def append[A](x: List[A], y: List[A]): List[A] = x match {
   case Nil => y
   case head :: Nil  => head :: y
   case head :: tail => head :: append(tail, y)
 }
 append(List(1,9,3), List(4,5,6))
```

Invertir una lista

Invertir una lista

```
def reverse(l: List[Int]) : List[Int] =
            if (l == Nil) l else reverse(l.tail) ::: List(l.head)


def reverse[A](x: List[A]): List[A] = x match {
   case head :: tail => append(reverse(tail), List(head))
   case Nil => Nil
  }

val a=List(1, 9, 3, 4, 5, 6)                    //> a  : List[Int] = List(1, 9, 3, 4, 5, 6)
reverse(a)                                      //> res1: List[Int] = List(6, 5, 4, 3, 9, 1)
```

Elevar al cuadro los elementos de una lista

Elevar al cuadro los elementos de una lista

```
val lista =List(1, 9, 3, 4, 5, 6)

Def cuadraLista (l:List[Int]):List[Int]={
        | if (l.isEmpty) Nil
        | else (l.head * l.head) :: cuadraLista(l.tail)
        | }
                                                    //> cuadraLista: (l: List[Int])List[Int]

        cuadraLista (lista)                         //> res1: List[Int] = List(1, 81, 9, 16, 25, 36)
```

Sumar dos números

```
object Exercises {
        def succ(n: Int) = n + 1
        def pred(n: Int) = n - 1

        def add(x: Int, y: Int): Int =
                if (y == 0) x else succ(add(x, pred(y)))

succ(3)                          //> res0: Int = 4
pred(1)                          //> res1: Int = 0
add(1,3)                         //> res2: Int = 4

 }
```

Suma de los elementos de una lista

```
object Exercises {
        def succ(n: Int) = n + 1
        def pred(n: Int) = n - 1

        def add(x: Int, y: Int): Int =
                        if (y == 0) x else succ(add(x, pred(y)))

 val lista =List(1, 9, 3, 4, 5, 6)

        def sum(x: List[Int]): Int =
                        if (x.isEmpty) 0 else add(x.head, sum(x.tail))

        add(8,9)                        //> res2: Int = 17
        sum(lista)                      //> res3: Int = 28
}
```

Suma de los elementos de una lista(patrones)

```
object Exercises {
        def succ(n: Int) = n + 1
        def pred(n: Int) = n - 1

         def add(x: Int, y: Int): Int = x match {
                case 0 => if(y == 0) 0 else add(y, x)
                case _ => if(x > 0) succ(add(pred(x), y))
                          else      pred(add(succ(x), y))
        }

         def sum(x: List[Int]): Int = x match {
                  case head :: Nil  => add(head, 0)
                  case head :: tail => add(head, sum(tail))  }
}
```

Longitud de una lista

```
object Exercises {
      def succ(n: Int) = n + 1
      def pred(n: Int) = n - 1

      def length[A](x: List[A]): Int =
      if (x.isEmpty) 0 else succ(length(x.tail))
}


def length[A](x: List[A]): Int = x match {
    case Nil => 0
    case head :: tail => succ(length(tail))
  }
```

Anadir elementos a una lista

```
val lista1 =List(1, 9, 3, 4, 5, 6)            //> lista1  : List[Int] = List(1, 9, 3, 4, 5, 6)

val lista2 =List(1, 9, 3, 4, 5, 6)            //> lista2  : List[Int] = List(1, 9, 3, 4, 5, 6)
def append[A](x: List[A], y: List[A]): List[A] = {
                def append2(x: List[A], acc: List[A]): List[A] =
                                        if (x.isEmpty) acc.head :: y
                                        else
                                        acc.head :: append2(x.tail, x.head :: acc)
                                        append2(x.tail, x.head :: Nil)
}                                                       //> append: [A](x: List[A], y: List[A])List[A]
                 append(lista1,lista2)                  //> res2: List[Int] = List(1, 9, 3, 4, 5, 6, 1, 9, 3, 4, 5, 6)

  def append1[A](x: List[A], y: List[A]): List[A] = x match {
                                case Nil => y
                                case head :: Nil  => head :: y
                                case head :: tail => head :: append1 (tail, y)
  }
                append1(lista1,lista2)                  //> res3: List[Int] = List(1, 9, 3, 4, 5, 6, 1, 9, 3, 4, 5, 6)
```

Concadenar elementos

```
def concat[A](x: List[List[A]]): List[A] = {
                        if (x.isEmpty)      Nil
                        else
                        append(x.head, concat(x.tail))
                }
 def concat[A](x: List[List[A]]): List[A] = x match {
                    case head :: Nil => head
                    case head :: tail => append(head, concat(tail))
 }
```

Valor máximo de una lista

```
def maximum(x: List[Int]): Int = {
        def maximum2(x: List[Int], max: Int): Int =
                if (x.isEmpty) max
                else
            maximum2(x.tail, if (x.head > max) x.head else max)
            maximum2(x.tail, x.head)
        }
def maximum(x: List[Int]): Int = x match {
         case head :: Nil => head
         case head :: tail => {
                val max_tail = maximum(tail)
                if(head > max_tail) head else max_tail
  }
 }
```

Devuelva dos veces una cadena, un “Int” si es un entero o “otro tipo”

```
val p: Any = "Hola"
p match {
        case s: String => s+s
        case s: Int => "Int"
        case _ => "Otro tipo" }

//      Res4: String = HolaHola
```

Currying

```
def unoporotro (x: Int) = (y: Int) => x * y
unoporotro (7)(8)                    //> res5: Int = 56

def resta (x: Int) = (y: Int) => x - y
resta  (8)(7)                        //> res6: Int = 1

def rest (x: Int, y: Int) = (x - y)
rest  (8,7)                          //> res7: Int = 1

val curryprueba = media(5, _: Double)
```

Currying

```
object holass extends App {
  def filter(xs: List[Int], p: Int => Boolean): List[Int] =
    if (xs.isEmpty) xs
    else if (p(xs.head)) xs.head :: filter(xs.tail, p)
    else filter(xs.tail, p)

  def modN(n: Int)(x: Int) = ((x % n) == 0)

  val nums = List(1, 2, 3, 4, 5, 6, 7, 8)

  println(filter(nums, modN(2)))          //> List(2, 4, 6, 8)
  println(filter(nums, modN(3)))          //> List(3, 6)
}
```

Evaluación de expresiones de izquierda a derecha

Por nombre

sumOfSquares(3, 2+2)
sumOfSquares(3, 4)
square(3) + square(4)
3 * 3 + square(4)
9 + square(4)
9+4*4
9 + 16
25

Por Valor

sumOfSquares(3, 2+2)
square(3) + square(2+2)
3 * 3 + square(2+2)
9 + square(2+2)
9 + (2+2) * (2+2)
9+4*(2+2)
9+4*4
9 + 16
25

Inserta un color en una posición del tablero dado

Inserta un color en una posición del tablero dado

```
def insertar (color: Int, pos: Int, tablero: List[Int]): List[Int] = {
        if (pos==1) color::tablero.tail
        else tablero.head::insertar(color, pos-1, tablero.tail)
        }
```

Inserta un color en una posición del tablero dado

```
def poner (col: Int, pos:Int, l:List[Int]): List[Int]= {
        if (l.isEmpty) Nil
        else if (pos==1) col::l.tail
        else l.head::poner (col, (pos-1), l.tail) }
//> poner: (col: Int, pos: Int, l: List[Int])List[Int]
val liston = List(0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
val t1:List[Int] = poner (7, 2, liston)
val t2:List[Int] =poner (7, 3, t1)
```

Ámbitos:
Reglas de evaluación de expresiones Scala

El funcionamiento de los ámbitos en Scala:

- Una invocación a una función crea un nuevo ámbito en el que se evalúa el cuerpo de la función. Es el ámbito de evaluación de la función.
- El ámbito de evaluación se crea dentro del ámbito en el que se definió la función a la que se invoca
- Los argumentos de la función son variables locales de este nuevo ambito que quedan ligadas a los parámetros que se utilizan en la llamada.
- En el nuevo ámbito se pueden definir variables locales.
- En el nuevo ámbito se pueden obtener el valor de variables del ámbito padre.

Ámbitos de las funciones

```
def f(x: Int, y: Int): Int = {
    val z = 5 x+y+z
    }
 def g(z: Int): Int = {
     val x = 10
    z+x }

f(g(3),g(5))
```

Ámbitos de las funciones

```
def f(x: Int, y: Int): Int = {
    val z = 5
    x+y+z
    }
 def g(z: Int): Int = {
     val x = 10
    z+x }

f(g(3),g(5))
```

2) g(5)
X=10
Z=5 =>Z=15

1) g(3)
X=10
Z=3 =>Z=13

3) f
X=13
Y=15
Z=5=>X+Y+Z=33

Ámbitos de las funciones

```
val x = 0
Val z = 100

Def f()= {
        val x = 10
        val y = x+20
        x+y+z
}

f()
X
```

Ámbitos de las funciones

```
val x = 0
val z = 100

Def f()= {
        val x = 10
        val y = x+20
        x+y+z
}

f()
X
```

X=0
Z=100

X=10
Y=30
f()=140

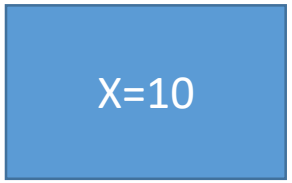
X=0

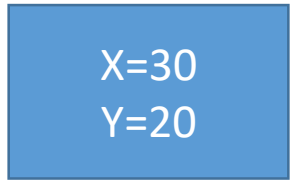
Ámbito de reasignación

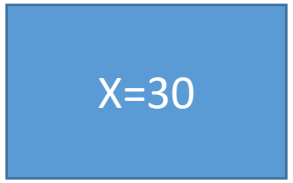
```
var x = 10

def change(y: Int)= {
        x = x+y
        x
}

change(20)
X
```

Ámbitos de las funciones

```
var x = 10

def change(y: Int)= {
        x = x+y
        x
}

change(20)
X
```







y si en lugar de var fuera val
¿qué pasaría ?

¿Como se evalúa la siguiente sentencia?

a ++: b

```
val a = Set(1,2,3,4)
val b = Set(3,4,5,6)
a ++: b
```

a) Es lo mismo que a++(b)
b) Se evalua de izquierda a derecha
c) Se evalua de derecha a izquierda
d) Da error de codigo

¿Cómo se evalúa la siguiente sentencia?

a ++: b

```
val a = Set(1,2,3,4)
val b = Set(3,4,5,6)
a ++: b
```

a) Es lo mismo que a++(b)

b) Se evalua de izquierda a derecha

c) Se evalua de derecha a izquierda (por los dos puntos)

d) Da error de codigo

Toma una lista de tests y un número n y que devuelva el número de tests de la lista que pasa el número n.

supongamos que los disponemos de los siguientes métodos:

mayorQue8(x) , par(x) , impar(x)

val listaTests = List(mayorQue8 _, par _, impar _)

numTests(listaTests, 12)
⇒2
numTests(listaTests, 3)
⇒1

Ejemplo pregunta

```
val lista: List[Int] = List(1, 2, 3, 4)

val primero = lista.head

val resto = lista.tail

val dobleLista = primero :: resto ::: lista
```

Ejemplo pregunta

```
val lista: List[Int] = List(1, 2, 3, 4)          //> lista: List[Int] = List(1, 2, 3, 4)

val primero = lista.head                          //> primero  : Int = 1

val resto = lista.tail                            //> resto  : List[Int] = List(2, 3, 4)

val dobleLista = primero :: resto ::: lista       //> dobleLista  : List[Int] = List(1,
                                                  //|  2, 3, 4, 1, 2, 3, 4)
```

Toma una lista de tests y un número n y que devuelva el número de tests de la lista que pasa el número n.

supongamos los disponemos de los siguientes métodos:

mayorQue8(x) , par(x) , impar(x)

val listaTests = List(mayorQue8 _, par _, impar _)

numTests(listaTests, 12)
⇒2
numTests(listaTests, 3)
⇒1