

Mémo Scala

Master DAC – Bases de Données Large Echelle
Mohamed-Amine Baazizi

baazizi@ia.lip6.fr

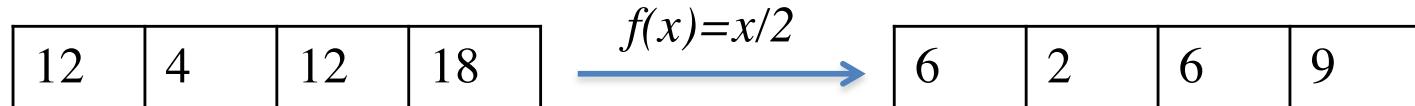
Octobre 2018

Remarque

Le but de cette introduction est de vous permettre de démarrer sur Spark. Il ne s'agit en aucun cas d'un cours exhaustif de Scala.

Map Reduce en bref

- Programmation fonctionnelle : fonctions d'ordre supérieur appliquant une fonction aux objets d'une collection C
- *Map* ($f: T \Rightarrow U$), *funaire* : appliquer f à chaque élément de C
T: Type de départ U: type retourné

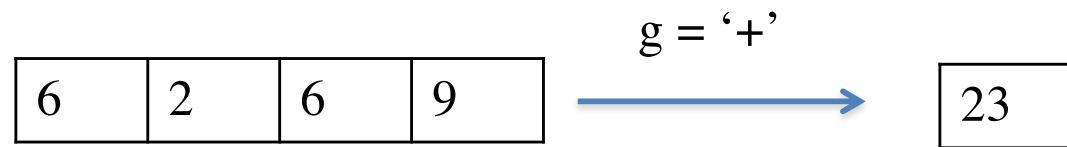


Propriétés : la dimension de C est préservée

le type en entrée peut changer

Map Reduce en bref

- Programmation fonctionnelle : fonctions d'ordre supérieur appliquant une fonction aux objets d'une collection C
- *Reduce* ($g: (T,T) \Rightarrow T$), g binaire
 - agrège les éléments de C deux à deux avec un opérateur binaire



$$(6 + (2 + (6 + 9))) = 23$$

Propriétés : réduit la dimension de n à 1
le type en sortie identique à celui en entrée

Scala en quelques mots

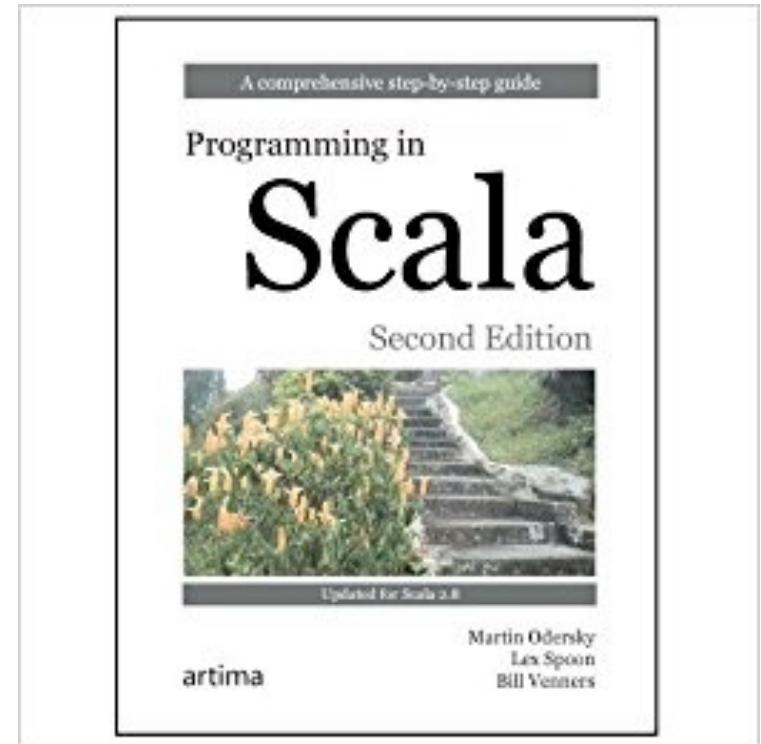
- Langage orienté-objet et fonctionnel à la fois
 - Orienté objet : valeur → objet, opération → méthode
Ex: l'expression 1+2 signifie l'invocation de la méthode '+ ' sur des objets de la classe Int
 - Fonctionnel
 - Les fonctions se comportent comme des valeurs : peuvent être retournées ou passées comme arguments
Ex: Map(x=>f(x)) avec f(x) = x/2
 - Les structures de données sont immuables (*immutable*) : les méthodes n'ont pas d'effet de bord, elles associent des valeurs résultats à des valeurs en entrée
Ex: c=[2, 4, 6] c.Map(x=>f(x)) produit une nouvelle liste [1, 2, 3]

Avantages de Scala

- Compatibilité avec Java
 - compilation pour JVM, types de base de Java (Int, float, ..)
- Syntaxe concise
 - Un prog. Scala = 1/2 lignes d'un prog. Java
- Haut niveau d'abstraction
 - possibilité de cacher des détails à l'aide d'interfaces
- Typage statique
 - inférence de types à partir des valeurs, utile pour éviter certaines erreurs pendant l'exécution

Plan

- Notions de base
- Types et opérations de base
- Structures de contrôle
- Types complexes
- Fonctions d'ordre supérieur



Référence bibliographique

M. Odersky, L. Spoon, B. Venners. *Programming in Scala*. 2nd Edition. 2012

https://booksites.artima.com/programming_in_scala_2ed

Ligne de commande

- Mode interactif

```
$ spark-shell  
...  
scala>
```

Manipulations de base

```
scala> 1+2  
res0: Int = 3  
  
scala> res0+3  
res1: Int = 6
```

res0	la valeur calculée
:Int	le type inféré
=3	la valeur calculée

Valeurs vs variables

- les valeurs sont immuables, i.e elles ne peuvent être modifiées

```
scala> val n=10
```

```
n: Int = 11
```

```
scala> n=n+1
```

```
<console>:12: error: reassignment  
to val  
      n=n+1  
      ^
```

```
scala> var m=10
```

```
m: Int = 10
```

```
scala> m=m+1
```

```
m: Int = 11
```

On ne peut réaffecter une nouvelle valeur à *n* car déclarée avec **val**

On peut incrémenter *m* car déclarée avec **var**

Définition des fonctions

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

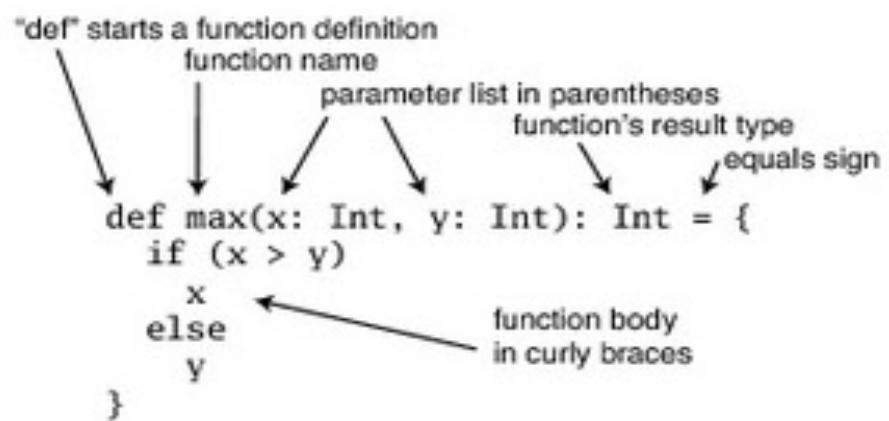
```
max2: (x: Int, y: Int)Int
```

```
scala> max(1,3)
```

```
res3: Int = 3
```

```
scala> max(max(1,2),3)
```

```
res6: Int = 3
```



Le type retour inféré automatiquement sauf pour les fonctions récursives
Type par défaut Unit : correspond à void en Java

```
scala> def bonjour() = println ("bonjour")  
bonjour: ()Unit
```

Types et opérations de base

Table 5.1 · Some basic types

Value type	Range
Byte	8-bit signed two's complement integer (-2 ⁷ to 2 ⁷ - 1, inclusive)
Short	16-bit signed two's complement integer (-2 ¹⁵ to 2 ¹⁵ - 1, inclusive)
Int	32-bit signed two's complement integer (-2 ³¹ to 2 ³¹ - 1, inclusive)
Long	64-bit signed two's complement integer (-2 ⁶³ to 2 ⁶³ - 1, inclusive)
Char	16-bit unsigned Unicode character (0 to 2 ¹⁶ - 1, inclusive)
String	a sequence of Chars
Float	32-bit IEEE 754 single-precision float
Double	64-bit IEEE 754 double-precision float
Boolean	true or false

- Opérateurs arithmétiques : + - * / %
- Opérateurs logiques : && || !
- Opérateurs binaires ...
- Conversions : `toInt` `toDouble` `toLowerCase` `toUpperCase`
 - à explorer en mode interactif

Conversions de type

- Plusieurs possibilités, à explorer en mode interactif

```
scala> val v = 124
v: Int = 124
scala> v.to
      Taper TAB
  to          toChar     toFloat    toLong     toShort
  toBinaryString toDegrees toHexString toOctalString toString
  toByte       toDouble    toInt      toRadians
```



```
scala> v.toInt
      Taper TAB
def.toInt: Int
```

Structures de contrôle

Paradigme fonctionnel, les structures retournent une valeur

- *if (cond) val
else alternative*

```
scala> val chaine = "abcde"
scala> val longueur =
           if (chaine.length %2 ==0) "pair"
             else "impair"
longueur: String = impair
```

- *While et for*
 - à éviter car style impératif. On privilégiera les *map* (cf. fonctions d'ordre supérieur)

Structures de contrôle

- *pattern match*
 - branchement conditionnel à n alternatives exprimés par un *pattern* dans la clause *case*

Syntaxe :

```
var match {  
    case val0 => res0  
    case val1 => res1  
    ...  
    case _ => res_defaut  
}
```

```
scala> def verif (age : Int) = age match {  
    case 25 => "argent"  
    case 50 => "or"  
    case 60 => "diamond"  
    case _ => "inconnu"  
}  
verif: (age: Int)String  
  
scala> verif(10)  
res7: String = inconnu
```

Types complexes

- le type tableau (Array)
 - séquence d'éléments (souvent du même type)
 - construction directe ou à partir de certaines fonctions comme le *split()*
 - accès indexé pour lecture ou écriture, indice 1^{er} élément = 0

```
scala> val weekend = Array ("sam", "dim")
weekend: Array[String] = Array(sam, dim)
```

```
scala> weekend(0)
res6: String = sam
```

```
scala> weekend(1)
res7: String = dim
```

0	1
"sam"	"dim"

Types complexes

- le type liste (List)
 - collection d’éléments (souvent du même type)
 - construction suivant différentes manières :
 - conversion d’un tableau
 - instantiation d’un objet List avec valeurs fournies
 - de manière récursive avec l’opérateur *cons* noté *elem::liste*

```
scala> val lweekend = weekend.toList  
lweekend: List[String] = List(ven, sam)
```

conversion

```
scala> val fruits = List("pomme", "orange", "poire")  
fruits: List[String] = List(pomme, orange, poire)
```

instanciation

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

concaténation

Types complexes

- Manipulation de listes
 - ajout en tête seulement (immuabilité)

```
scala> 4 :: unAtrois
res13: List[Int] = List(4, 1, 2, 3)
```

```
scala> val quatreAun = 4 :: unAtrois.reverse
quatreAun: List[Int] = List(4, 3, 2, 1)
```

- concaténation à l'aide de :: - l'ordre interne est préservé

```
scala> val quatreAcinq = 4 :: 5 :: Nil
quatreAcinq: List[Int] = List(4, 5)
```

```
scala> val unAcinq = unAtrois ::: quatreAcinq
unAcinq: List[Int] = List(1, 2, 3, 4, 5)
```

Types complexes

- dés-imbrication de listes : la méthode *flatten*

```
scala> val nestd_unAinq = List(unAtrois, quatreAinq)
nestd_unAinq: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
```

```
scala> nestd_unAinq.flatten
res19: List[Int] = List(1, 2, 3, 4, 5)
```

Pourquoi
le type
Any ?

```
unAhuit: List[List[Any]] = List(List(List(1, 2, 3), List(4, 5)), List(6, 7))
```

```
scala> unAhuit.flatten
res24: List[Any] = List(List(1, 2, 3), List(4, 5), 6, 7)
```

Types complexes

- **Tuples**
 - Collection d’attributs relatifs à un object (cf. modèle rel.)
 - Accès indexé avec `._index` où `index` commence par 1
 - structure immuable, construits souvent à partir de sources externes (ex. fichier csv)

```
scala> val tuple = (12, "text", List(1,2,3))
tuple: (Int, String, List[Int]) = (12, text, List(1, 2, 3))
```

```
scala> tuple._1 = 13
<console>:12: error: reassignment to val
          tuple._1 = 13
                      ^
```

Types complexes

- **Tuples et pattern matching**
 - possibilité de reconnaître la forme des tuples et d'enclencher un traitement spécifique en utilisant des variables

```
scala> val listeTemp = List((7,2010,4,27,75), (12,2009,1,31,78))
listeTemp: List[(Int, Int, Int, Int, Int)] = List((7,2010,4,27,75),
(12,2009,1,31,78))

scala> listeTemp.map{
    case(sid,year,month,value,zip)=>(year,value)
}
res0: List[(Int, Int)] = List((2010,27), (2009,31))
```

Types complexes

- Tableaux associatifs (map)
 - ensemble de paires (clé, valeur) - unicité de clé – clé et valeur de type quelconque mais fixés une fois pour toute
 - possibilité d'insertion et de mise à jour de nouvelles paires

```
scala> var capital = Map("US" -> "Washington", "France" -> "Paris")
capital...
```

```
scala> capital("US")
res2: String = Washington
```

```
scala>capital += ("US" -> "DC", "Japan" -> "Tokyo")
```

```
Map(US -> DC, France -> Paris, Japan -> Tokyo)
```

Types complexes

- Classes
 - Conteneurs pour objets ayant les mêmes attributs
 - *class MaClasse (nom: String, num: Int)*
{ //attributs et méthodes – partie optionnelle }

```
scala> class Mesure(sid:Int, year:Int, value:Float)
```

```
defined class Mesure
```

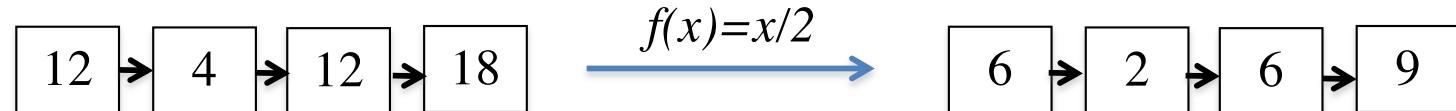
```
scala> listeTemp.map{case(sid,year,month,value,zip)=>new  
Mesure(sid,year,value)}
```

```
res2: List[Mesure] = List(Mesure@364c93e6, Mesure@66589252)
```

Fonctions d'ordre supérieur

- *map*

Map (f: T=>U), f uneire : applique f à chaque élément de C



la dimension de C est préservée mais le type en entrée peut changer

```
scala> def divide(n:Int) = n/2
succ: (n: Int)Int
```

```
scala> val l= List(12,4,12,18)
```

```
scala> l.map(x=>divide(x))
res1: List[Int] = List(6, 2, 6, 9)
```

Fonctions d'ordre supérieur

- $l.flatMap(f)$: équivalent de $l.map(f)$ suivi de $flatten$

```
scala> def succ(n:Int) = n+1
succ: (n: Int)Int
```

```
scala> nestd_unAinq
res38: List[List[Int]] = List(List(1, 2, 3), List(4, 5))
```

```
scala> val deuxAsix = nestd_unAinq.flatMap(succ)
deuxAsix: List[Int] = List(2, 3, 4, 5, 6)
```

- $l.foreach(f)$: applique f à chaque élément sans retourner de valeur

```
scala> quatreAinq.foreach(println)
4
5
```

Fonctions d'ordre supérieur

- *filter(cond)*
 - cond retourne un booléen et permet de sélectionner les éléments de la liste sur laquelle filter est appliqué

```
deuxAsix: List[Int] = List(2, 3, 4, 5, 6)
```

```
scala> deuxAsix.filter(x=>x%2 ==0)
```

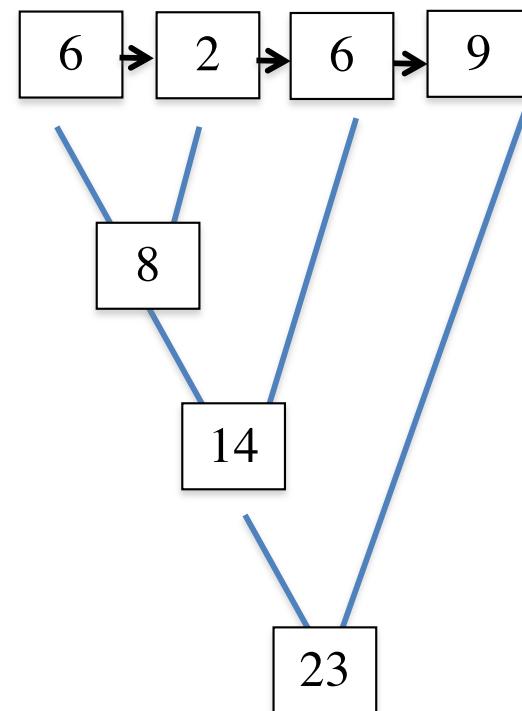
```
res46: List[Int] = List(2, 4, 6)
```

la condition s'exprime avec =>
et signifie :
retourner x si x est multiple de 2

Fonctions d'ordre supérieur

- Réduction : *reduce*
 - $l.reduce(g: (T,T) \Rightarrow T)$: applique g sur les éléments de l

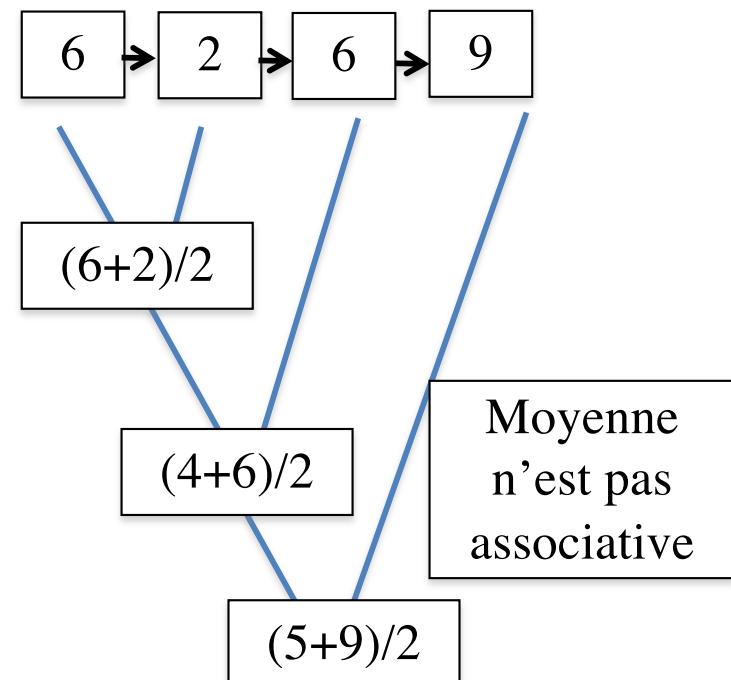
```
scala> def g(a:Int, b:Int) = {  
    println(a+"\t"+b)  
    a+b  
}  
  
scala> val l=List(6,2,6,9)  
  
scala> l.reduce((a,b)=>g(a,b))  
6  2  
8  6  
14 9  
res17: Int = 23
```



Fonctions d'ordre supérieur

- *reduce* ne marche que si g est associatif!
 - $l.reduce(g: (T,T) \Rightarrow T)$: applique g sur les éléments de l

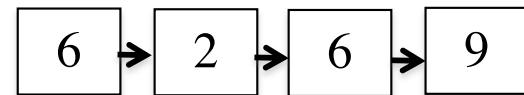
```
scala> def moyAB(a:Int, b:Int)=  
 {   println(a+"\t"+b)  
   (a+b)/2  
 }  
  
scala> val l=List(6,2,6,9)  
  
scala> l.reduce((a,b)=>moyAB(a,b))  
6  2  
4  6  
5  9  
res19: Int = 7
```



Fonctions d'ordre supérieur

- *reduce* ne marche que si g est associatif!
 - $l.reduce(g: (T,T) \Rightarrow T)$: applique g sur les éléments de l

```
scala> val l=List(6,2,6,9)  
  
scala> val sum = l.reduce((a,b)=>a+b)  
  
scala> val moy = sum/l.count()
```



pour calculer le moyenne, il faut calculer la somme puis diviser sur le taille de la liste!

Bien entendu, on peut utiliser la fonction `avg()` prédéfinie.