TD 1 : Premiers pas en Scala

Université Paris Diderot – Master 2

21 septembre 2015

1 Simply Scala

Le premier TD est disponible entièrement en ligne à :

`http://www.simplyscala.com/fr`

A Tutoriel I

A.1 Comment évaluer les fragments de code.

Tous les exemples de ce tutoriel peuvent être évalués en cliquant simplement sur leur code source.

Vous pouvez rentrer votre propre code en le tapant à l'intérieur de la boîte située tout en haut de cette page. Le code sera envoyé et évalué par une simple pression sur la touche "Entrée". Vous pouvez taper des fragments de code d'une ligne de cette façon. Cependant, si vous commencez un bloc de code à l'aide d'un délimiteur d'ouverture (par exemple, une accolade ouvrante), le code ne sera envoyé que lorsque la version fermante de ce délimiteur sera rencontrée (dans notre exemple, une accolade fermante) et que le curseur est placé à la fin du code source. Ceci permet d'écrire un code source de plusieurs lignes et de l'éditer librement. Il est toujours possible de cliquer sur le bouton d'évaluation ou sur "Ctrl"+"Entrée" pour forcer l'envoi du code. Des exemples venus d'autres endroits peuvent ainsi être insérés par le biais de copier/coller standard. La taille maximale du texte est limitée à 2000 caractères.

Le code qui a déjà été envoyé peut être rappelé en utilisant la touche "Ctrl" en combinaison des touches directionnelles "haut" et "bas". Vous pouvez éditer ce code à l'aide des touches directionnelles "haut" et "bas" et l'envoyer de nouveau. Ainsi, vous pouvez essayer un exemple, le rappeler, le modifier, essayer de nouvelles idées et vérifier comment certains aspects de la syntaxe.

Bouton "Reset" : ce bouton met à zéro votre espace utilisateur sur le serveur. Le serveur sauvegarde tous vos résultats intermédiaires de façon à ce que vous puissiez les utiliser dans de prochains calculs. Cela peut prendre un peu de temps. Si vous n'avez pas besoin de ces résultats, cliquez sur "Reset". La réponse du serveur à vos requêtes arrivera alors plus vite. Les entrées précédentes peuvent toujours être rappelées à l'aide des touches directionnelles.

A.2 Expressions

Les expressions arithmétiques fonctionnent comme on s'y attend. On trouve les opérateurs habituels et leurs priorités relatives standards s'appliquent. Les parenthèses peuvent être utilisées pour contrôler l'ordre des applications.

```
1+2
3+4*(2-3)
23%5 // Reste de la division entière
3.5*9.4+6/4
```

Scala fournit différents types de nombre. "3.5" est un "Double" alors que "6" est un "Int". Notez que, dans le dernier cas, le résultat de la division entière est tronquée en une valeur entière. Scala peut coercer les valeurs dans le type requis par le contexte dans une expression mêlant nombres entiers et nombres flottants, quand c'est possible.

Le résultat d'une expression peut être stocké dans une variable. Les noms de variables commencent par une lettre qui peut être suivie de chiffres ou de lettres. Les chaînes "Golf1", "helpLine" et "Res4" sont toutes trois des exemples de noms de variables. Le résultat d'une expression peut être associé à un nom de variable. Cette association est réalisée par le mot-clé "var" ou le mot-clé "val". "val" est utilisé pour une association à établir une fois pour toute et qui ne peut pas être modifiée par la suite. Pour le moment, vous pouvez utiliser "var". Pourquoi cette distinction entre "var" et "val" ? "val" définit une valeur non modifiable et comme vous allez l'apprendre plus loin, ceci est caractéristique du style de programmation fonctionnel.

```
val pixel=34+5
var height=pixel+4
println(height)
```

Maintenant, essayez :

```
pixel=10 // ceci produit une erreur car on essaie de modifier une variable déclarée par "val".
height+=4
println(height)
```

Vous pouvez accéder aux précédents résultats en utilisant les variables "res1", "res2", etc. Les commentaires peuvent être rajoutés à la fin du code à l'aide du délimiteur "//" ou sur plusieurs à l'aide d'un couple "/* et */". Tous les commentaires sont ignorés.

```
/*
  Exemple utilisant "res", and ce type de commentaires.
*/
res0 + " Scala"  // plus de commentaires, ici.
```

On trouve aussi un ensemble complet d'opérateurs de manipulation bit à bit.

```
3&2 // et logique
1|2 // ou logique
1^2 // xor logique
1<<2 // décalage à gauche
-24>>2 // décalage à droite avec préservation du signe.
-14>>>2 // décalage à droite ajoutant des zéros à gauche.
```

Pour le moment, vous avez déjà en main une puissante calculatrice mais il serait utile de pouvoir programmer le contrôle de flot pour traiter des calculs qui se répètent par exemple.

A.3 Types de base

Scala a un ensemble de types de base prédéfinis. Ils sont résumés dans ce qui suit :

```
Byte      Entier 8-bit signé en complément à 2 (-128 à 127 inclus)
Short     Entier 16-bit signé en complément à 2 (-32,768 à 32,767 inclus)
Int       Entier 32-bit signé en complément à 2 (-2,147,483,648 à 2,147,483,647 inclus)
Long      Entier 64-bit signé en complément à 2 (-2^63 à 2^63-1 inclus)
Float     Flottant 32-bit IEEE 754 en précision simple
Double    Flottant 64-bit IEEE 754 en précision double
Char      Caractère Unicode 16-bit non signé
String    Une séquence de caractères Unicode
Boolean   true/false
```

A.4 If else

"if (cond) else" est la première des différentes structures de contrôle de flot, elle permet de choisir de faire un calcul plutôt qu'un autre suivant une certaine condition. Il y a un certain nombre d'opérateurs de comparaison à partir desquels on peut construire des expressions booléennes. En voici quelques-uns :

```
1>2 // plus grand que
1<2 // plus petit que
1==2 // égal à
1>=2 // plus grand ou égal à
1!=2 // différent de
1<=2 // plus petit ou égal à
```

Avec la commande "if" de Scala, si la condition est vérifiée alors l'expression avant le "else" est évaluée sinon c'est l'expression après le "else" qui l'est. Dans tous les cas, on obtient une valeur.

```
if(1>2) 4 else 5 // greater than
if (1<2) 6 else 7 // less than
val try1=if (1==2) 8 else 9 // equals
val isBook = 6>=3
val price=16
val vol=10
val sale=if (isBook)price*vol else price/vol sale
```

Vous pouvez avoir besoin de combiner des conditions atomiques. Il y a deux opérateurs qui servent à cela. "&&" signifie "ET". "||" signifie "OU". Ils combinent des valeurs booléennes tandis que "&" et "|" travaillent sur les représentations binaires.

```
val isBook = 6>=3
val price=16
val vol=10
val sale=if (((isBook)&&(price>5))||(vol>30))price*vol else price/vol
sale
```

A.5 while

"while (cond) block/exp" vous permet de répéter l'évaluation d'un bloc de code ou d'une commande tant qu'une condition est vérifiée.

```
var total=18
while(total < 17) total+=3
```

"do block/exp while (cond)" vous permet de répéter l'évaluation d'un bloc de code ou d'une commande tant qu'une condition est vérifiée mais celle-ci est évaluée après chaque itération.

```
var total=18
do{
  total+=3
}while (total < 17)
```

Notez que dans ce dernier cas, la variable "total" vaut "21" à la fin du calcul alors qu'elle valait "18" dans l'exemple précédent du "while". Voici maintenant, un autre exemple qui calcule le plus grand diviseur commun.

```
// find the greatest common divisor
var x = 36
var y = 99
while (x != 0) {
  val temp = x
  x = y % x
  y = temp
}
println("gcd is",y)
```

A.6 for

"for (range) block/exp" vous permet de répéter l'évaluation d'un bloc de code en itérant sur toutes les valeurs d'une collection (une collection peut être une liste, un segment d'entiers naturels, un ensemble fini).

```
for(i <- 1 to 4) println("hi five")
```

La variable "i" prend successivement les valeurs entières comprises entre 1 et 4. Si vous voulez exclure le dernier élément de l'itération, vous pouvez utiliser le mot-clé "until" :

```
for(i <- 1 until 4) println(i)
```

Les itérations multidimensionnelles sont élégamment pris en compte en fournissant une liste de déclarations d'itérateurs :

```
for(i <- 1 until 4 ; j <- 1 to 3) println(i,j)
```

Comme dit plus haut, "for" peut être utilisé pour itérer sur n'importe quelle sorte de collection. Par exemple, une chaîne de caractère est une collection de caractères donc "for" permet d'itérer sur chacun d'eux.

```
for(c<-"hello")println(c)
```

A.7 Littéraux

Les littéraux vous permettent de définir des valeurs de types de base. Ce sont à peu près les mêmes que Java.

A.8 Entiers

Il y a 4 types d'entiers "Int", "Long", "Short" et "Byte". Vous pouvez écrire des littéraux représentés dans différentes bases (décimale, hexadécimale et octale). On signale la base désirée à l'aide des premiers caractères du littéral.

Décimal (base 10) : Tout nombre qui commence par un chiffre différent de zéro.

```
17
298
```

Hexadécimal (base 16) : Tout nombre qui commence par 0x ou 0X et est suivi des chiffres de "0" à "9", "a" à "f" ou "A" à "F".

```
0x23 //hex = 35 dec
0x01FF //hex = 511 dec
0xcb17 //hex = 51991 dec
```

Octal (base 8) : Tout nombre qui commence par le chiffre "0" et est suivi des chiffres de "0" à "7".

```
023 // octal = 19 dec
0777 // octal = 511 dec
0373 // octal = 251 dec
```

Par défaut, ces valeurs entières sont de type "Int". Vous pouvez les forcer à avoir le type "Long" en rajoutant la lettre "l" ou "L".

```
0XFAF1L // hex long = 64241
035L
```

Vous pouvez affecter les littéraux à des variables de type "Short" ou "Byte". Cependant, la valeur doit être dans le domaine approprié pour ce type.

```
val abyte: Byte = 27
val ashort: Short = 1024
val errbyte: Byte = 128 // Erreur - pas dans le segment "-128" à "127"
```

A.9 Nombre à virgule flottante

Les littéraux à virgule flottante sont des nombres contenant le symbole ".". Ils doivent commencer par un chiffre différent de 0 et peuvent être suivis d'un "E" ou d'un "e" qui sert à préfixer l'exposant indiquant la puissance de 10 à utiliser.

```
9.876
val tiny = 1.2345e-5
val large = 9.87E45
```

Par défaut, les littéraux à virgule flottante sont de type "Double" mais vous pouvez les forcer à être du type "Float" en rajoutant la lettre "f" ou "F". On peut rajouter "d" ou "D" à la fin d'un littéral à virgule flottante si on le souhaite.

```
val sma = 1.5324F
val ams = 3e5f
```

A.10 Caractère

Les littéraux de caractères sont spécifiés par n'importe quel caractère Unicode entre apostrophes.

```
val chr = 'A'
```

Vous pouvez aussi spécifier sa valeur de différentes façons.
En octal : Un nombre octal entre '\0' et '\377'.

```
val chr = '\101' // code de "A"
```

En Unicode : Un nombre hexadécimal entre '\u0000' et '\uFFFF'.

```
val chra = "\\u0041 est un A"
val chre = "\\u0045 est un E"
```

Pour finir, on peut aussi spécifier quelques littéraux à l'aide de séquences d'échappement introduites par un symbole "". Reportez vous à la référence pour une liste complète.

A.11 Chaîne de caractères

Un littéral de chaîne de caractères est une séquence de caractères entourée de guillemets.

```
val helloW = "hello world"
val someEsc = "\\\"\'"
```

Scala offre une syntaxe spéciale pour éviter ces échappements multiples de caractères. Si vous commencez et terminez une chaîne par un triple guillemets alors tous les caractères comme les retour à la ligne, les points d'interrogation et les caractères spéciaux sont traités comme tous les autres.

```
println("""Welcome to Simply Scala.
Click 'About' for more information.""")
```

A.12 Booléen

Le type des booléens a deux valeurs et ses littéraux sont "true" et "false" :

```
val isBig = true
val isFool = false
```

A.13 Fonctions

Les fonctions vous permettent de définir des calculs pouvant être répétées. Une fonction est définie à l'aide du mot-clé "def". L'exemple qui suit crée une fonction qui retourne la valeur maximale de deux entiers. Une fonction retourne une valeur d'un certain type. Cependant, certaine fonction ne retourne pas de valeur informative. Dans ce cas, leur type de retour est "unit".

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

Le nom de la fonction, "max" dans notre exemple, suit le mot-clé "def". Puis, viennent les paramètres avec leur type associé entre parenthèse. Une annotation de type est ajoutée après ces paramètres, juste après les deux-points, il s'agit du type de retour de la fonction. Après le signe'=', on trouve la définition du corps de la fonction. Dans notre cas, il s'agit d'un bloc de code que l'on reconnaît à la présence d'accolades l'englobant. Une fois que vous avez défini une fonction, vous pouvez l'utiliser et l'appelant avec des paramètres appropriés :

```
max(6,7)
```

Les fonctions peuvent s'appeler récursivement. Les fonctions récursives forment une alternative pour contrôler les itérations. Dans la fonction suivante, on calcule le plus grand diviseur commun sans utiliser explicitement de cellules mémoires pour y stocker les résultats intermédiaires.

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

Comparez ce programme au version précédente écrite à l'aide boucle "while".

```
gcd(96,128)
```

A.14 Tout est objet

Scala est un langage orienté objet. Cela signifie, comme pour tous les autres langages orientés objets, que les programmes sont composés d'objets ayant un état et cet état est manipulable ou observable par le biais de méthodes, les fonctions des objets qui leur servent d'interface. Les objets sont définis à l'aide de hiérarchies de classe. Quand vous définissez une classe, vous définissez aussi un nouveau type d'objet. Ce nouveau type d'objet a le même statut que les types prédéfinis comme "Int" ou "Double", toutes les valeurs de ces types sont de type "Object". L'intérêt de cette uniformisation va apparaître un peu tard dans ce tutoriel.

Vous pouvez commencer par définir un objet qui représente un point.

```
class Point {
var x=0
  var y=0
}
```

Une classe est une définition abstraite pour cet objet dans le sens où elle décrit un générateur d'objets concrets. Une instance de cette classe d'objets peut être créée à l'aide du mot-clé "new".

```
val p=new Point
```

Les variables définies à l'intérieur de cet objet peuvent être accédées à l'aide de la notation pointée.

```
p.x=3
p.y=4
```

Vous pouvez récupérer l'état de la même façon.

```
println(p.x,p.y)
```

Il n'est pas très efficace de mettre les champs d'une nouvelle instance de point à jour un à un. En rajoutant des paramètres à la définition de la classe, on peut construire une valeur dont les champs ont directement les valeurs initiales désirées.

```
class Point(ix:Int,iy:Int){
  var x=ix
  var y=iy
}
```

On peut ensuite créer un point initialisé correctement :

```
val p=new Point(3,4)
```

Maintenant, supposons que nous voulons ajouter deux points pour créer un nouveau point de façon à réaliser une sorte d'addition entre vecteurs. On peut rajouter une méthode pour cela :

```
class Point(ix:Int,iy:Int){
  var x=ix
  var y=iy
  def vectorAdd(newpt:Point):Point={
    new Point(x+newpt.x,y+newpt.y)
  }
}
```

Deux points que l'on peut additionner peuvent alors être créés :

```
val p1=new Point(3,4)
val p2=new Point(7,2)
val p3=p1.vectorAdd(p2)
println(p3.x,p3.y)
```

À ce stade, ceci ressemble très fortement à ce qu'un programmeur Java attend. Cependant, il serait plus naturel d'être "p1+p2". En Scala, ceci est possible. Les noms des méthodes sont composés de la quasi-totalité des symboles alphanumériques. (Quelques combinaisons sont réservées et vous obtiendrez une erreur si vous essayez de les utiliser.) On peut donc réécrire la classe précédente en faisant usage du symbole "+". On rajoute aussi une méthode nommée "-" :

```
class Point( ix:Int,iy:Int){
  var x=ix
  var y=iy
  def +(newpt:Point):Point={
    new Point(x+newpt.x,y+newpt.y)
  }
  def -(newpt:Point):Point={
    new Point(x-newpt.x,y-newpt.y)
  }
  override def toString="Point("+x+","+y+")"
}
val p1=new Point(3,4)
val p2=new Point(7,2)
val p3=new Point(-2,2)
val p4=p1+p2-p3
println(p4.x,p4.y)
```

Avec cette façon de faire, on peut exprimer des calculs vectoriels dans un langage très naturel et facile à lire.

En Scala, il existe une simplification supplémentaire pour créer des classes grâce un mécanisme de "hiérarchies de classes définies par cas". La classe "point" peut être écrite ainsi à l'aide d'une "classe-cas" :

```
case class Point(x:Int,y:Int){
  def +(newpt:Point)=Point(x+newpt.x,y+newpt.y)
  def -(newpt:Point)=Point(x-newpt.x,y-newpt.y)
  override def toString="Point("+x+","+y+")"
}
val p1=Point(3,4)
val p2=Point(7,2)
val p3=Point(-2,2)
p1+p2-p3
```

Notez que la définition explicite des champs de la classe n'est pas obligatoire avec ce type de déclaration. Vous pouvez aussi noter que le mot-clé "new" n'est plus nécessaire pour créer une nouvelle instance. Le compilateur Scala détecte

automatiquement le besoin d'instanciation et insert une instruction de création d'instance pour vous. Vous pouvez aussi remarquer que les accolades ne sont plus présentes dans les définitions de deux méthodes. Elles ne sont pas nécessaires puisque le corps de la méthode est une simple expression et non pas un bloc de commandes à effectuer. Ceci est d'ailleurs vrai pour tout type de définition, pas seulement les définitions de méthodes. Pour finir, remarquez que le type de retour des méthodes n'est pas précisé : le compilateur Scala intègre un moteur d'inférence de type capable de le déduire lui-même à partir du corps de la méthode.

Ceci est un simple avant-goût de ce qui fait de Scala un langage beaucoup plus concis que Java. Plus vous apprendrez Scala, et plus vous réussirez à exprimer des idiomes de programmation de façon concise dans ce langage.

Tout est un objet. Pour cette raison, vous devez vous demander pourquoi le calcul décrit plus haut ne suit pas le style plus orienté objet :

```
p1.+(p2.-(p3))
```

C'est en fait grâce à une fonctionnalité sympathique de Scala qui aide à améliorer la clarté et l'uniformité de votre code. Vous pouvez en effet supprimer les parenthèses et les points car le compilateur de Scala déduit lui-même la structure des appels de méthodes que vous effectuez. Cette syntaxe a été minutieusement conçue pour vous permettre l'implémentation de Langages Spécifiques à un Domaine (Domain Specific Languages, DSLs) directement en Scala.

Ainsi, tous les objets, mêmes les valeurs numériques, ne sont que des objets avec des méthodes. D'ailleurs, vous pouvez réciproquement appliquer la méthode "+" au nombre "1" avec la notation pointée :

```
(1).+(2)
```

Contrairement à Java, tous les types de base sont aussi des types d'objet qui peuvent être utilisés et étendus comme n'importe quel autre type.

Notez que le premier jeu de parenthèse autour du "1" est nécessaire pour supprimer une ambiguïté syntaxique : en effet, "1." est un littéral de type "Double". Sans les parenthèses, le résultat de cette expression serait de type "Double" et non "Int". Essayez de faire ce changement !

A.15 switch - Analyse de motifs

Vous êtes peut-être déjà familier avec la construction "switch" associée à des éléments "case" que l'on trouve dans de nombreux langages de programmation pour autoriser le branchement multiple sur une valeur. En Scala, ce concept est étendu de façon à offrir toute la généralité des motifs dits algébriques par l'utilisation du mot-clé "match".

Avant toute chose, notons qu'une analyse classique du type "switch" peut être représentée facilement à l'aide d'un match :

```
def  decode(n:Int){
  n match {
    case 1 => println("One")
    case 2 => println("Two")
    case 5 => println("Five")
    case _ => println("Error")
  }
}
decode(2)
```

Le symbole "=>" est utilisé pour séparer le motif de l'analyse de l'expression ou du bloc à évaluer. Le symbole "_" est utilisé en Scala pour représenter le cas par défaut (correspondant au complémentaire des cas qui le précèdent). Comme toutes les expressions du langage un "match" construit une valeur, qui peut naturellement être utilisée comme un résultat intermédiaire d'une expression englobante. La fonction de l'exemple précédent peut donc s'écrire de façon plus concise :

```
def  decode(n:Int){
  println(n match {
```

```
    case 1 => "One"
    case 2 => "Two"
    case 5 => "Five"
    case _ => "Error"
    }
  )
}
decode(3)
```

Contrairement à la construction “switch” traditionnelle de Java, l’association représentée par l’analyse par cas précédente peut aisément être inversée :

```
def  encode(s:String){
  println(s match {
    case "One" => 1
    case "Two" => 2
    case "Five" => 5
    case _ => 0
    }
  )
}
encode("Five")
```

Dans le prochain exemple, on définit un arbre binaire dont les noeuds internes et les feuilles sont annotés par des valeurs. On peut observer que l’analyse de motifs permet de déterminer le type d’un noeud et d’associer des noms à ces paramètres nommés par les lettres minuscules k, l, r et v. Ces dernières sont appelées “variables de motif”. Un motif peut aussi être formé à l’aide de constantes que l’on reconnaît lexicalement par le fait qu’elles débutent par une majuscule ou qu’elles sont représentées par un symbole littéral. Dans ce cas, la valeur analysée est comparée directement à cette constante.

```
abstract class TreeN
case class InterN(key:String,left:TreeN,right:TreeN) extends TreeN
case class LeafN(key:String,value:Int) extends TreeN

def find(t:TreeN,key:String):Int={
   t match {
     case InterN(k,l,r) => find((if(k>=key)l else r),key)
     case LeafN(k,v) => if(k==key) v else 0
  }
}
// create a binary tree
val t=InterN("b",InterN("a",LeafN("a",1),LeafN("b",2)),LeafN("c",3))
/*       [b]
         / \
       [a] c,3
       / \
     a,1 b,2
*/
```

Notez l’utilisation de la construction “case class” qui permet de créer facilement un arbre binaire pour notre test. Maintenant, vous pouvez essayer la fonction “find”.

```
find(t,"a")
find(t,"c")
```

Vous pourriez avoir envie d'encapsuler ceci dans une classe d'arbre binaire, en y incluant les méthodes d'insertion, de suppression et de recherche d'entrées.

Supposez que, pour une raison quelconque, vous désirez cacher la clé "c" durant la recherche. Une simple modification de la fonction "find" est nécessaire et elle illustre l'utilisation d'une constante dans un motif.

```
def find(t:TreeN,key:String):Int={
   t match {
     case InterN(k,l,r) => find((if(k>=key)l else r),key)
     case LeafN("c",_) => 0
     case LeafN(k,v) => if(k==key) v else 0
   }
}
```

Notez l'utilisation du symbole "_" pour filtrer n'importe quelle valeur. Souvenez vous aussi que les branches d'une analyse sont considérées dans l'ordre d'apparition dans le code source.

A.16 Typage statique et inférence de type

Scala est un langage statiquement typé, toutes les variables et les fonctions ont des types qui sont totalement définis au moment de la compilation. Utiliser une variable ou une fonction d'une façon inappropriée produit donc une erreur avant même l'exécution du programme. Cela signifie que le compilateur peut vous aider à découvrir des erreurs de programmation. Cependant, vous avez du remarquer que les exemples précédents ne font apparaître que très peu d'annotations de types. En effet, dans la plupart des cas, Scala est capable de déterminer le type d'une variable à partir de la façon dont on l'utilise.

Par exemple, si vous écrivez "val x = 3" alors le compilateur va inférer que "x" doit avoir le type "Int" parce que "3" est un symbole littéral entier. Dans les quelques situations où le compilateur n'est pas capable de deviner le type que vous avez en tête, il va produire une erreur. Dans ces cas-là, une simple annotation de type permet de faire accepter votre programme (si il est véritablement bien typé).

En général, vous devez spécifier les types des paramètres des fonctions. Par contre, le compilateur peut très souvent inférer le type de retour, que l'on omet donc la plupart du temps. La définition de fonctions récursives est une exception à cette règle : dans tous les cas, il faut spécifier leurs types de retour.

L'inférence de type réduit considérablement la quantité d'annotations de typage à écrire, ce qui augmente de façon importante la clarté du code. C'est cette inférence de type qui donne le sentiment que Scala est dynamiquement typé, alors qu'il l'est statiquement en vérité.

A.17 Apprendre Scala et ce tutoriel

L'objectif de ce tutoriel est de vous donner un rapide tour d'horizon des fonctionnalités basiques de Scala par la pratique. Il se focalise sur les choses essentielles que vous devez connaître pour commencer à écrire des programmes sans rentrer dans les détails du langage. Pour comprendre Scala plus en profondeur, cliquez sur le lien "Learn More Scala". Vous trouverez alors des livres et d'excellents supports d'apprentissage pour étendre vos compétences. Vous pouvez toujours revenir sur cette page pour essayer des exemples de programmes.

Ou bien, vous pouvez continuer la seconde partie de ce tutoriel en cliquant sur l'onglet intitulé "Tutorial-II".

B Tutoriel II

B.1 Les fonctions sont aussi des objets

En Scala, tout est objet et ceci inclut les fonctions. Elles peuvent être passées en arguments, retournées par d'autres fonctions ou stockées dans des variables. Cette fonctionnalité de Scala permet de résoudre des problèmes de programmation très courants d'une façon concise et élégante. Elle autorise aussi des structures de contrôle de flot très flexibles. Par exemple, les "Actors" de Scala font une utilisation poussée de ce mécanisme pour permettre la programmation concurrente. Un autre exemple, plus accessible pour cette introduction, concerne les manipulations

de liste. Considérons le problème : comment extraire les entiers impairs d'une liste ? Voici une entrée possible de ce problème :

```
val lst=List(1,7,2,8,5,6,3,9,14,12,4,10)
```

Les trois méthodes de liste "head", "tail" et " : :" seront utilisées dans les exemples suivants. À partir de ces dernières, vous allez voir comment construire de nouvelles fonctions très utiles travaillant sur les listes. La méthode "head" retourne le premier élément de la liste (si il existe), la méthode "tail" retourne une liste sans son premier élément (si il existe) et la méthode " : :" retourne une nouvelle liste avec l'élément fourni mis en tête d'une liste donnée elle-aussi en argument. Voici une solution à notre problème :

```
def odd(inLst:List[Int]):List[Int]={
  if(inLst==Nil) Nil
  else if(inLst.head%2==1) inLst.head::odd(inLst.tail)
  else odd(inLst.tail)
}
odd(lst)
```

Il est aussi très simple d'écrire une fonction qui retourne une liste des entiers pairs.

```
def even(inLst:List[Int]):List[Int]={
  if(inLst==Nil) Nil
  else if(inLst.head%2==0) inLst.head::even(inLst.tail)
  else even(inLst.tail)
}
even(lst)
```

Cependant, si on passait une fonction qui encapsule la condition de filtrage en argument, on pourrait obtenir une solution plus générale à notre problème. On définit donc tout d'abord une condition de filtrage :

```
def isodd(v:Int)= v%2==1
```

et ensuite on modifie la fonction de filtrage d'une liste en lui rajoutant un paramètre correspondant à la condition de filtrage. Bien que paramètre soit une fonction, il est considéré par Scala comme un objet quelconque. Notez la forme de la déclaration de type. Le type de l'argument est une fonction qui attend un entier et produit un booléen. Seules les fonctions de ce type peuvent être passées en argument. Dans le corps de la fonction de filtrage, la condition de filtrage "cond" est utilisée comme n'importe qu'elle autre fonction.

```
def filter(inLst:List[Int],cond:(Int)=>Boolean):List[Int]={
  if(inLst==Nil) Nil
  else if(cond(inLst.head)) inLst.head::filter(inLst.tail,cond)
  else filter(inLst.tail,cond)
}
filter(lst,isodd)
```

Bien que le cas des nombres pairs puisse être traité de la même façon, une fonction anonyme permet d'obtenir une version encore plus concise. Pour cela, le mot-clé "def" et l'identificateur de la fonction sont supprimés et la définition de la fonction est passée directement en argument. Notez l'utilisation du symbole "$=>$" pour séparer la liste des paramètres de la fonction de son corps.

```
filter(lst,(v:Int)=> v%2==0)
```

Voici la solution finale à notre problème. Comparez la à la solution initiale.

B.2 Un petit goût de programmation générique

Supposons maintenant que nous souhaitions créer un filtre pour des listes dont les éléments sont de type "Double". Vous pourriez aller remplacer "Int" par "Double" dans la définition de fonction précédente et appeler "filterD" la fonction obtenue. Avec cette méthode, pour tout nouveau type d'élément, vous auriez à effectuer le même exercice et vous finiriez par avoir plusieurs versions du même programme.

En Scala, vous pouvez éviter cet effort de duplication en utilisant une variable de type à la place des occurrences des types précédents. Au moment de l'utilisation de la fonction sur un type particulier de listes, le compilateur substituera à cette variable de type le type réel des éléments, au cas par cas.

Ici l'identificateur "T" est utilisé pour représenter la variable de type à la place de "Int" dans la fonction "filter". De plus, le nom de la fonction est annoté par "[T]" pour indiquer que la fonction "filter" est générique et paramétrée par la variable de type "T".

```
def filter[T](inLst:List[T],cond:(T)=>Boolean):List[T]={
  if(inLst==Nil) Nil
  else if(cond(inLst.head)) inLst.head::filter(inLst.tail,cond)
  else filter(inLst.tail,cond)
}
filter(lst,(v:Int)=> v%2==0)
```

Cette nouvelle fonction travaille aussi bien sur des listes de "Double" :

```
val lstd=List(1.5,7.4,2.3,8.1,5.6,6.2,3.5,9.2,14.6,12.91,4.23,10.04)
filter(lstd,(v:Double)=> v>5)
```

que sur des listes de chaînes de caractères :

```
val lsts=List("It's","a","far","far","better","thing","I","do","now")
filter(lsts,(v:String)=> v.length>3)
```

En fait, vous pouvez vérifier dans la documentation de référence de la bibliothèque standard de Scala que les objets de type "list" ont une fonction "filter" définie par défaut. On aurait donc pu écrire :

```
lsts.filter((v:String)=> v.length>3)
```

B.3 Compléments sur les "fonctions comme objets"

Manipuler des ensembles créés par "compréhension", c'est-à-dire créés par application d'une fonction à tous les membres d'une collection, est une façon très efficace de compacter le code source.

Très souvent, il est utile de passer des fonctions très simples en argument. Scala offre des raccourcis très pratiques pour cela. Vous avez déjà vu que l'inférence de type peut aussi aider. Dans l'exemple précédent, on peut encore simplifier notre programme parce que le compilateur est capable de deviner que l'argument de notre fonction est du type "String". On peut alors supprimer les parenthèses et l'annotation de type :

```
lsts.filter(v=>v.length>3)
```

Comme les opérateurs binaires sont fréquemment utilisés, Scala propose une notation encore plus concise pour eux. Par exemple, la fonction anonyme "(x, y) $=>$ x + y" peut s'écrire "$+$". De la même façon, "v $=>$ v.Method" peut être remplacé par "*.Method*". *Le symbole "*" représente un trou à combler ce qui vous évite d'avoir à inventer des noms. Notre exemple se réécrit donc :

```
lsts.filter(_.length > 3)
```

En lisant des programmes écrits en Scala, vous vous apercevrez que ces notations courtes sont très souvent utilisées. Cependant, il faut parfois utiliser les formes longues lorsque le compilateur n'arrive pas à inférer correctement la fonction que vous avez en tête.

Voici quelques exemples supplémentaires de manipulation de listes avec des fonctions en argument. Ils sont tirés du manuel de référence.

```
flatMap
lsts.flatMap(_.toList)
```

Vous pouvez remarquer que "flatMap" prend une liste et utilise la fonction fournie pour créer une nouvelle liste. Dans ce cas, il aplatit la liste de mots en caractères et concatène les sous-listes pour obtenir le résultat.

B.3.1 sort

Les éléments d'une liste peuvent être triés dans l'ordre ascendant en utilisant la méthode "sort".

```
lsts.sort(_<_)
```

ou bien dans un ordre descendant si on le précise.

```
lsts.sort(_>_)
```

ou bien en ignorant la casse des caractères.

```
lsts.sort(_.toUpperCase<_.toUpperCase)
```

En passant la fonction appropriée, on peut engendrer une famille d'algorithmes de tri sans avoir à réécrire l'algorithme lui-même. C'est une façon très agréable de faire varier le comportement d'une méthode sans avoir à la réécrire entièrement.

B.3.2 fold

foldLeft et foldRight permettent de combiner les éléments d'une liste en utilisant une fonction arbitraire. Ce processus peut commencer soit à partir du début (left) de la liste ou de sa fin (right). Il faut aussi fournir une valeur initiale à combiner avec le premier élément de la liste à considérer.

```
val lst=List(1,7,2,8,5,6,3,9,14,12,4,10)
lst.foldLeft(0)(_+_)
```

La fonction "foldLeft" utilise la fonction "$+$" pour additionner la valeur initiale "0" avec l'élément le plus à gauche "1". Ensuite, "7" est additionné au résultat et ainsi de suite jusqu'à ce que toute la liste soit parcourue.

B.3.3 removeDuplicates

Supposons que vous vouliez déterminer l'ensemble des lettres qui apparaissent dans une liste de mots :

```
val lstw=List("Once","more","unto","the","breach")
lstw.flatMap(_.toList).map(_.toUpperCase).removeDuplicates.sort(_<_)
```

Dans cet exemple, "flatmap" aplatit tous les mots en une liste de lettres. "map" convertit toutes ces lettres en majuscule. Toutes les occurrences multiples sont alors supprimées. Le résultat est finalement trié par ordre croissant.

B.4 n-uplets

Il est souvent utile d'écrire des fonctions qui calculent plusieurs valeurs à la fois. Vous pouvez bien sûr créer une classe au cas par cas pour stocker ces valeurs mais c'est assez lourd. Pour résoudre ce problème, Scala autorise d'ailleurs la création de classes anonymes en tout point du programme. On peut aussi utiliser des n-uplets. Syntaxiquement, un n-uplet est un ensemble de valeurs entourées de parenthèses. Un n-uplet peut contenir un ensemble de valeurs de types hétérogènes.

```
(3,'c')
```

Un n-uplet est un objet donc on peut accéder à ses composantes en utilisant la notation pointée. Ces champs ont des noms prédéfinis formés d'un caractère de soulignement "_" suivi d'un indice de position dans le n-uplet.

```
(3,'c')._1
(3,'c')._2
```

Il est cependant plus fréquent d'utiliser le mécanisme de déconstruction des n-uplets. Par exemple,

```
val (i,c)=(3,'a')
```

Nous allons voir comment créer maintenant une table de fréquences pour les lettres de notre liste de mots.

L'approche consiste à aplatir "lstw", la convertir en majuscule, puis trier la liste de caractères obtenue. Ensuite, à l'aide d'un combinateur "fold", on compte le nombre d'occurrences de chaque lettre rencontrée. Vous avez déjà vu comment "fold" permet de combiner une valeur d'entrée avec les éléments successifs d'une liste. Dans notre cas, la valeur véhiculée par "fold" est la table de fréquence. Pour combiner la lettre rencontrée et cette table de fréquence, c'est très simple : si la lettre est égale à la précédente lettre rencontrée alors on incrémente le compteur de cette dernière dans la table, sinon on rajoute une nouvelle entrée dans la table pour la nouvelle lettre considérée.

La table des fréquences va être représentée par une liste de couples contenant une lettre et un entier représentant sa fréquence. On créé d'abord la liste triée de caractères :

```
val ltrs=lstw.flatMap(_.toList).map(_.toUpperCase).sort(_<_)
```

Maintenant, on effectue un "fold". La valeur initiale est une table de fréquence vide, représentée par une liste vide. On doit ensuite fournir une fonction qui effectue la combinaison de la table des fréquences et d'une valeur de type "Char", le caractère suivant de la liste :

```
ltrs.foldLeft(List[(Char,Int)]()){
  case ((prevchr,cnt)::tl,chr) if(prevchr==chr) =>(prevchr,cnt+1)::tl
  case (tbl,chr) => (chr,1)::tbl
}
```

Pour comprendre ce fragment de programme quelques éléments supplémentaires de la syntaxe de Scala sont nécessaires.

Tout d'abord, une séquence de "case"s entre accolades peut être utilisée pour représenter une fonction définie par cas. C'est un sucre syntaxique pour une fonction dont le corps est une analyse de motifs :

```
(a,b) => (a,b) match {case ...}
```

Ensuite, dans notre exemple, le mot-clé "case" est utilisé pour déconstruire la table de fréquence (qui est une liste de couples) et la lettre en cours d'inspection. Le motif'(prevchr,cnt) : :tl' capture une liste avec un élément en tête et une queue, et associe un nom à chacune de ces composantes. "chr" fait référence au second argument passé à la fonction utilisée par "fold", la lettre en cours d'inspection.

On peut aussi compléter un motif d'une analyse à l'aide d'une garde. La garde commence avec le mot-clé "if" et peut contenir une expression booléenne arbitraire. Dans le cas de notre exemple, la garde est utilisée pour restreindre le motif à ne capturer que les cas où le caractère précédent est égal au caractère en cours d'inspection.

Maintenant, nous avons toutes les informations nécessaires sur la syntaxe de Scala pour décrypter ce programme : "Pour tout caractère de la liste d'entrée, si il y a déjà une entrée dans la table des fréquences, remplacer cette entrée de la table en incrémentant le compteur. Dans l'autre cas, rajouter simplement une nouvelle entrée dans la table avec un compteur valant 1."

Bien sûr, vous pourriez aussi utiliser une approche beaucoup plus proche de ce que l'on écrirait en Java :

```
def IFreqCount(in:List[Char]):List[(Char,Int)]={
  var Tbl=List[(Char,Int)]()
  if(in.isEmpty)Tbl
  else{
    var prevChr=in.head
    var nxt=in.tail
    var Cnt=1
    while (!nxt.isEmpty){
```

```
      if(nxt.head==prevChr)Cnt+=1
      else {
        Tbl=(prevChr,Cnt)::Tbl
        Cnt=1
        prevChr=nxt.head
        }
      nxt=nxt.tail
      }
    (prevChr,Cnt)::Tbl
  }
}
IFreqCount(ltrs)
```

Vous allez trouver que cette façon de traiter les fonctions comme des objets est très utile pour un grand nombre de tâches de programmation. Par exemple, pour passer des fonctions de rappel (callbacks) à des entrées-sorties événementielles, pour passer des tâches à effectuer à des acteurs dans un environnement de calculs concurrents ou encore pour effectuer de la répartition de charge de calculs. Très souvent, cela permet d'obtenir un programme plus concis.

Vous verrez aussi que l'idée d'utiliser des fonctions qui ne modifient pas les données mais en créent de nouvelles simplifient aussi les applications concurrentes. Cela vaut le coup d'apprendre à les manipuler.

B.5 Prochaines étapes

Dans ce tutoriel, seule une minuscule partie des bibliothèques de Scala a été présentée. Vous êtes déjà familier avec beaucoup de constructions importantes du langage Scala et vous pouvez donc commencer à écrire des programmes sophistiqués dans ce langage. Si vous êtes déjà un programmeur Java, vous serez en mesure de réutiliser toutes les bibliothèques de votre environnement Java existant car Scala et Java interopèrent de façon transparente.

Un bon livre est essentiel. "Programming in Scala" de Martin Odersky, Lex Spoon et Bill Venners est une bonne référence pour continuer. Cliquez sur "Learn More Scala" pour consulter cette référence et d'autres livres. Il existe d'excellents supports d'apprentissage pour Scala et des explications sur la marche à suivre pour télécharger Scala et installer un IDE. Amusez-vous bien !

Et, bien sûr, vous pouvez toujours revenir sur Simply Scala pour essayer des exemples...