Diverses utilisations de la programmation fonctionnelle en Scala

Jacques Noyé

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

24 janvier 2017

1 De Haskell à Scala, traitements simples sur les listes

1.1 Méthodes

Traduire en Scala, sous forme de méthodes, les fonctions définies de la manière suivante en Haskell. Vous n'avez pas besoin de curryfier les méthodes (utiliser une unique liste de paramètres). À la place de `myAppend`, utiliser l'opérateur prédéfinie de concaténation des listes `:::`).

```
myLength :: [a] -> Int
myLength [] = 0
myLength (_:xs) = 1+myLength xs

myReverse :: [a] -> [a]
myReverse (x:xs) = myAppend (myReverse xs) [x]
myReverse xs = xs

myReverse1 :: [a] -> [a]
myReverse1 xs = myReverse2 xs []
                where myReverse2 :: [a] -> [a] -> [a]
                      myReverse2 [] a = a
                      myReverse2 (x:xs) a = myReverse2 xs (x:a)
```

Pour tester : vous pouvez simplement créer un projet `FP`, mettre vos définitions dans un fichier, par exemple `fp.scala` à la racine, et charger ce fichier dans REPL à l'aide de la commande `load` :

```
scala> :load fp.scala
```

1.2 Fonctions

Reprendre l'exercice précédent en définissant des fonctions dans un trait paramétré par le type des éléments.

Donner deux versions alternatives de `myReverse1`. Dans la première, implémenter `myReverse2` sous la forme d'une méthode ; dans la deuxième, sous la forme d'une fonction.

2 Programmation par évènements

Sur Campus, le fichier `CounterJava.zip` contient un petit projet en Java qui permet d'associer un compteur à une interface graphique minimale avec deux boutons pour incrémenter et décrémenter le compteur et afficher sa valeur.

2.1 Passage à Scala

Traduire le projet Java `CounterJava` en un projet Scala `Counter` sachant que toutes les classes et interfaces Java importées dans le premier projet peuvent être importées dans le deuxième (les interfaces Java sont alors vues comme des traits Scala) et utilisées de la même manière.

Il y a deux subtilités à ne pas manquer :

— les méthodes d'incrémentation et de décrémentation doivent bien être les seules méthodes permettant de modifier la valeur du compteur ;

— le nom du compteur et de sa fenêtre graphique sont les mêmes.

2.2 Amélioration

On aimerait simplifier l'utilisation de la méthode `addActionListener` en lui passant en paramètre, plutôt qu'un objet de type `ActionListener`, une fonction de type `ActionEvent => Unit` afin de pouvoir écrire, par exemple :

```
buttonPlus.addActionListener((_:ActionEvent) => counter.increment())
```

Autrement dit, l'idée est de rendre explicite le fait qu'un *event listener* n'est rien d'autre qu'un *gestionnaire d'évènement*[1], soit une fonction qui prend un évènement en entrée et le traite.

Utiliser à cette fin une conversion implicite (à simplement placer dans la classe `GuiCounter`) qui convertit un gestionnaire d'évènement tel qu'il a été défini ci-dessus en un objet de type `ActionListener`.

3 Introduction de nouvelles structures de contrôle

La combinaison des facilités syntaxiques de Scala et de la programmation fonctionnelle permet de créer de nouvelles structures de contrôle.

Définir une méthode `tantque` implémentant l'équivalent d'une boucle `while` soit, informellement, tant que le premier paramètre est vrai le deuxième paramètre est évalué.

Par exemple :

```
val a = Array(1, 2, 3)
var i = 0
tantque (i < a.length) { a(i) = a(i) + 1; i = i + 1 }
for(i <- 0 to 2) print(a(i))
```

Vous allez avoir besoin d'utiliser une méthode curryfiée et un passage de paramètres par nom.

4 Passage de paramètres nommés et définition de valeurs par défaut

En Scala, il est aussi possible de nommer les paramètres et ainsi de s'affranchir de leur ordre. On peut ainsi écrire :

1. Un *event handler* en anglais.

```
scala> def concat1(x: String, y: String) = x + y
add: (x: Int, y: Int)Int

scala> concat1(x = "X", y = "Y")
res1: String = XY
```

Il est aussi possible de définir des valeurs par défaut des paramètres :

```
scala> def concat2(x: String, y: String = "Y") = x + y
concat2: (x: String, y: String)String

scala> concat2("X")
res2: String = XY
```

Ou encore, en mélangeant les deux :

```
scala> def concat3(x: String = "X", y: String) = x + y
concat3: (x: String, y: String)String

scala> concat3(y = "Y")
res3: String = XY
```

Cela dit, mélanger ces facilités avec la curryfication peut être hasardeux. Considérons les définitions suivantes :

```
def invert(z: String)(y: String = "Y", x: String = "X") = x + y + z
val invertxy = invert("Z") _
```

Quel est le résultat de l'exécution des expressions suivantes, et pourquoi ?

```
invertxy(x = "X")
invertxy(x = "X", y = "Y")
invertxy(v1 = "X", v2 = "Y")
```

Par contre, la curryfication peut être utilisée pour fournir une valeur ou des valeurs par défaut. Si le dernier paramètre d'une méthode currifyée est déclarée `implicit` et que cette méthode est appelée sans ce dernier paramètre, le compilateur va essayer de voir s'il n'existe pas une valeur *implicite* visible de ce paramètre (une valeur du bon type, déclarée `implicit`).

Est-ce que cette explication vous suffit pour définir une variante de `concat2` ?

4.1 Application partielle

Est-ce que la fonction (au sens mathématique) `cos` fournie par le paquetage `scala.math` est une méthode ou une fonction (au sens du langage Scala) ?

Définir dans un objet `FP` (pour *functional programming* ou *fixpoint*) une méthode `selfCompose` qui étant donnée une fonction f sur les nombres flottants doubles, un entier n positif et un nombre flottant double retourne f^n.

Utiliser cette méthode pour calculer le point fixe attractif de cosinus, c'est-à-dire la valeur de x_0 point fixe de cosinus ($cos(x_0) = x_0$) telle que toute suite $u_{n+1} = cos(u_n)$ converge vers x_0 (utiliser une *worksheet*).

Redéfinir `selfCompose` afin de généraliser le type flottant en un type `T` quelconque. Vérifier qu'elle s'applique toujours bien à cosinus.

Définir dans un trait `FP`, en utilisant la méthode précédente, une *fonction* `selfCompose`, qui ne prend que la fonction et l'entier en paramètre. Vérifier à nouveau qu'elle s'applique toujours bien à cosinus (par exemple en utilisant une classe anonyme héritant du trait).

Étudier les différents styles d'écriture de cette fonction :

1. définition explicite des paramètres et de leur type ;
2. définition implicite du paramètre `x` du résultat ;
3. définition du type de la fonction plutôt que de ses paramètres.