Université de Bordeaux
Année 2016-2017

Mastère 2
T.A.P

TD n°5 - Types de données et programmation fonctionnelle

Exercice 1: Map-Reduce

Les méthodes de l'interface `IEnumerable<T>` en C# (cf. `http://msdn.microsoft.com/en-us/library/9eekhta0.aspx`) permettent de mettre en place relativement facilement des algorithmes de types *map-reduce*. La fonction suivante permet de créer un graphe aléatoire :

```
Random rnd = new Random();                                  // Initialize RNG
List<Vertex<String>> graph = new List<Vertex<String>>(); // Create graph

for (int i = 0; i < 5; i++)                                 // Create vertices
  graph.Add(new Vertex<String>(i, "Vertex_" + i.ToString()));

for (int i = 0; i < 5; i++) {                               // Add edges once
  Vertex<String> v = graph[i];
  for (int j = 0; j < 5; j++)
    v.AddNeighbor(graph[rnd.Next(5)]); }

int index = rnd.Next(5);                                    // Add edges again
for (int i = 0; i < 5; i++)                                 // to a central vertex
  graph[i].AddNeighbor(graph[index]);
```

Pour supplémenter à quelques manques de la bibliothèques standard, le code source fournit une implémentation de la méthode `Flatten` (qui concatène une liste de listes en une unique liste) et de la méthode `ForEach` (qui exécute une fonction sur chaque élément de la liste).

1. Écrire une fonction permettant d'afficher la liste des arêtes du graphe, sommet par sommet, uniquement à l'aide des méthodes de `IEnumerable`.

2. Écrire une fonction permettant de calculer, pour chaque sommet, le nombre d'arêtes pointant vers ce sommet, à l'aide des méthodes de `IEnumerable`, en utilisant un algorithme de type map-reduce.

Exercice 2: Des interrogations plein les yeux

Le Java Collections Framework (`https://docs.oracle.com/javase/8/docs/technotes/guides/collections/overview.html`) contient un ensemble de classes génériques permettant de gérer des ensembles d'objets à travers des `Collection`s : `Set`, `List`, `Map` ... Il apporte aussi une classe `Collections` (`https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html`) contenant un ensemble de

méthodes statiques génériques. Beaucoup de ces méthodes comportent des types avec des *wildcards*, comme par exemple la méthode `copy` :

```java
// Copies all of the elements from one list into another.
static <T> void copy(List<? super T> dest, List<? extends T> src)
```

1. En toute généralité, quel lien y a t'il en Java entre deux objets des types suivants :

   - `List<Integer>` et `List<Number>` ;
   - `List<Integer>` et `ArrayList<Integer>`.

   En particulier, est-il possible d'assigner l'un dans l'autre et réciproquement ? Vérifier en écrivant le code correspondant.

Au vu de ces exemples, la relation "est-un" ne semble pas être préservée quand on passe d'un objet à sa collection : même si une class `Orange` hérite d'une classe `Fruit`, il n'y a aucune relation entre une `List<Orange>` et une `List<Fruit>`.

2. Donner un exemple montrant en quoi ce choix est restrictif.

La bibliothèque standard Java apporte une solution à ce problème à travers des indications de généricité comme des *wildcards*. Pour une variable possédant un tel type, il existe un principe général caractérisant les méthodes que l'on peut ou pas lui appliquer tout en préservant la sûreté de typage. Ce principe est appelé le *Get and Put principle*[1] :

- use an `extends` wildcard when you only `get` values out of a structure,
- use a `super` wildcard when you only `put` values into a structure, and
- don't use a wildcard when you both `get` and `put`.

Pour chacune des citations suivantes[2], écrire un exemple en Java qui ne compile pas, et expliquer pourquoi.

3. A propos des `extends` :

   ▷ For example the type `List<? extends Number>` is often used to indicate read-only lists of `Number`s. This is because one can get elements of the list and statically know they are `Number`s, but one cannot add `Number`s to the list since the list may actually represent a `List<Integer>` which does not accept arbitrary `Number`s.

[1] Tiré de *Java Generics and Collections*, M. Naftalin & P. Wadler
[2] Tiré de `http://www.cs.cornell.edu/~ross/publications/tamewild`

4. A propos des `super` :

   > ▷ Similarly, `List<? super Number>` is often used to indicate write-only lists. This time, one cannot get `Number`s from the list since it may actually be a `List<Object>`, but one can add `Number`s to the list.

5. Écrire une fonction Java `flatten` qui concatène une liste de listes en une unique liste.

   Note : le but est de s'appliquer à la rendre aussi générique que possible.

Exercice 3: Visiteurs du soir

Le pattern *visiteur* est un motif de programmation fondamental qui permet d'étendre l'ensemble des méthodes utilisables par toute une hiérarchie de classes d'objets. Dans cet exercice, nous allons utiliser le mécanisme des délégués (*delegate*) du langage C# afin d'implémenter ce motif en appliquant une technique de programmation fonctionnelle.

Considérons pour l'instant l'interface suivante :

```
public interface IFigure { String GetName (); };
```

Le code source contient un objet `SimpleFigure` qui satisfait à l'interface `IFigure`.

1. Implémenter une signature pour une figure composite `CompositeFigure` qui satisfasse l'interface `IFigure` et qui soit composée d'une liste de `SimpleFigure`.

Pour mettre en place le motif *Visiteur*, il est nécessaire que l'interface `IFigure` contienne une méthode nommée traditionnellement `Accept`. En programmation objet, cette méthode prend en argument un objet implémentant une interface nommée *Visiteur*, et contenant une méthode `Visit`. Ici, nous allons utiliser une tactique différente où le *Visiteur* est seulement un *delegate* implémentant la fonction `Visit` :

```
public delegate T VisitorFun<V, T>(V f);
```

2. Rajouter la ligne suivante dans l'interface `IFigure`, afin de préparer l'interface à recevoir des visiteurs fonctionnels :

```
T Accept<T>(VisitorFun<IFigure, T> v);
```

3. Implémenter la fonction `Accept` pour `SimpleFigure` et `CompositeFigure`.

   Rappel : le visiteur d'une figure simple doit simplement s'appliquer à la figure, le visiteur d'une figure composite doit s'appliquer récursivement à tous les éléments du composite ainsi qu'à lui-même.

Appliquons maintenant cette construction en construisant un visiteur qui construit la liste des noms (obtenus à travers `getName()`) de tous les éléments composant une figure, concaténés. Pour cela, il faut que le visiteur fasse un parcours de l'arbre composite en concaténant tous les noms dans une chaîne de caractères qu'il aura enfermé.

4. Écrire une fonction `MakeVisitorFun` renvoyant un délégué de type :

   `VisitorFun<IFigure, String>`,

   et qui soit un visiteur des objets satisfaisant à `IFigure`. Cette fonction doit se présenter de la manière suivante :

```
public static VisitorFun<IFigure , String> MakeNameFigureVisitorFun ( ) {
  string _state = "" ;
  return fig ⇒ // ... TODO
}
```

Remarque : en C#, il est possible de tester dynamiquement l'égalité de types en utilisant la construction suivante :

```
if (obj is SimpleFigureF) ...
```