Université de Bordeaux
Année 2019-2020

Mastère 2
T.A.P

TD n°3 - Programmation fonctionnelle et 1ère classe

Exercice 1: Interfaces fonctionnelles Java

Les *interfaces fonctionnelles* ont été introduites en Java 8, afin, entre autres, de pouvoir typer les lambdas-expressions. La liste de ces interfaces est accessible à `https://docs.oracle.com/javase/8/docs/api/java/util/function/package-summary.html`. Chacune de ces interfaces contient au moins une méthode particulière qui correspond à la fonction principale représentée par l'interface : `apply` pour `Function<T,R>`, `test` pour `Predicate<T>` ...

Afin d'expérimenter avec ces interfaces, on propose de travailler sur les données suivantes :

```
class Client {
    public final String first_name;
    public final String last_name;
    public double size;
    public Client(String f, String l, double h);
    public void increase();  // double internal size
    public void roar();      // speaks its name
}
List<Client> clients = Arrays.asList(
                           new Client("Allo",  "Saurus",  8.5),
                           new Client("Diplo", "Docus",   27),
                           new Client("Draco", "Rex",     3),
                           new Client("Mono",  "Clonius", 5),
                           new Client("Toy",   "Sarus",   9.3)
                           );
```

L'idée de cet exercice consiste à écrire des fonctions permettant de faire des requêtes sur cette base de données de clients.

1. Écrire une méthode statique `andAll` qui prend une liste de `Predicate<T>` en paramètre, et renvoie un prédicat qui est vrai lorsque tous les prédicats de la liste le sont.
2. Appliquer cette méthode pour construire le prédicat filtrant les clients d'une taille ≥ 5, avec un prénom commençant par "D" ou "T", et un nom finissant par "us".
   Remarque : utiliser à bon escient les méthodes `startsWith` et `endsWith` de la classe `String`.
3. Écrire une méthode statique `doAll` qui prend une liste de `Consumer<T>` en paramètre, et renvoie un consommateur qui applique tous les élements de la liste en séquence.
4. Appliquer cette méthode pour appliquer sur chacun des clients la liste des méthodes suivantes : `toString`, `roar`, `increase` et enfin `roar`.

Exercice 2: Objets-enregistrements

Les *enregistrements* sont des types de données capables de contenir plusieurs éléments de types différents. Néanmoins, il est nécessaire de créer explicitement de type de données :

```
type ratio = {
  num   : int;
  denum : int};;

let x = {num = 7; denum = 3 };;
x.num;; (* int = 7 *)
```

Supposons vouloir créer un type de données permettant de représenter des nombres complexes, sous la forme d'enregistrements. De plus, on aimerait que l'on puisse agir avec ces enregistrements comme avec des objets, en les munissant d'une opération d'addition et d'une opération de multiplication (qui s'écrive par exemple `z1.add z2`).

1. Proposer un type d'enregistrement permettant de représenter un tel type.
2. Implémenter un constructeur pour un tel type, *i.e.* une fonction prenant en paramètres deux nombres flottants et renvoyant un nombre complexe.

Plutôt que de passer par des enregistrements pour créer des objets, il est possible de créer directement des fonctions avec un comportement ressemblant à l'objet. Le code suivant en Python représente des nombres complexes sous la forme d'une fonction :

```
def new_complex(x,y):
    def anon(*args):
        if (not args): # return if args is the empty list
            return
        mess = args[0]
        if (mess == "show"):
            return (x,y)
        else:
            ox = args[1][0]; oy = args[1][1]
            if (mess == "add"):
                return new_complex(x+ox, y+oy)
            elif (mess == "mul"):
                return new_complex(x*ox-y*oy, x*oy+y*ox)
    return anon

c1 = new_complex(1,1)
c2 = c1("add", (1,1))
c3 = c2("mul", (2,2))
print(c3("show"))
```

3. Quelles difficultés poserait l'écriture d'une telle fonction en OCaml ?

▷ Les listes du langage OCaml sont des types de données permettant de regrouper des éléments du même type. Une liste vide est représentée par `[ ]`. Une liste à plusieurs éléments est représentée par `[` $\langle a_1 \rangle$ `;` $\langle a_2 \rangle$`; ... ;` $\langle a_n \rangle$ `]`.Pour construire une liste dans une expression, il est possible soit :

— d'adjoindre un élément ⟨x⟩ en tête de la liste ⟨c⟩ par l'opération ⟨x⟩::⟨c⟩

— de concaténer deux listes ⟨r⟩ et ⟨s⟩ par l'opération ⟨r⟩@⟨s⟩

Le module `List` (dont la description est disponible à l'adresse `http://caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`) fournit un certain nombre d'accesseurs (en particulier `List.hd` pour obtenir le 1er élément et `List.tl` pour le reste de la liste) et de fonctions sur les listes.

Exercice 3: Listes d'associations génériques

Une *liste d'associations* est une liste de couples (clé,valeur) générique. De telles listes permettent de stocker des informations dans les valeurs, en les codant à l'aide de clés.

1. Exprimer le type de données générique OCaml associé à une telle liste.
2. Écrire une fonction générique `list_remove` qui supprime le ou les éléments d'une liste d'associations égaux à une clé donnée.

```
list_remove 2 [(1,"un"); (2,"deux"); (2,"zwei")]
- : string list = [ (1,"un") ]
```

3. Écrire une fonction générique `list_map` qui, étant donnée une fonction `f` de transformation, transforme tous les éléments de la liste.

```
list_map fst [("sinus",sin); ("cosinus",cos)]
- : string list = [ "sinus"; "cosinus" ]
```

4. Comment implémenter la fonction de concaténation de deux listes d'associations, de manière à ce que l'algorithme soit récursif terminal ?