Programmation IV	Série 4	5 avril 2006

Exercice 1

Définissez la fonction `any` qui prend en argument une liste et un prédicat et retourne vrai ssi il existe au moins un élément dans la liste pour lequel ce prédicat est vrai. Votre fonction aura le profil suivant :

```
def any[A](p: A => boolean)(l: List[A]): Boolean
```

Définissez ensuite la fonction `every` qui teste si *tous* les éléments d'une liste satisfont un prédicat. (Note : nous vous demandons d'utiliser le filtrage de motifs dans les deux fonctions).

Exercice 2

Les *listes associatives* sont des listes de paires dont le premier élément représente une clef, et le second représente la valeur associée à cette clef. Une clef n'apparaît jamais plus d'une fois dans une telle liste. Par exemple, la liste

```
List(Pair(1965, "Jean"), Pair(1976, "Marcel"), Pair(1982, "Josette"))
```

associe un nom à une année de naissance.

Écrivez une fonction `lookup` qui prend en argument une liste associative et une clef, et qui retourne l'élément associé à cette clef dans la liste. Utilisez la fonction `error` pour signaler une erreur au cas où l'élément n'existe pas. Votre fonction `lookup` aura le profil suivant :

```
def lookup[A,B](lst: List[Pair[A,B]], key: A): B
```

Conseil : utilisez le filtrage de motif vu au cours pour obtenir une solution élégante et concise.

Exercice 3

Les listes associatives de l'exercice précédent sont un exemple de structures de données associatives, qui lient une valeur à une clef. Le désavantage principal de ces listes est que la fonction de recherche (`lookup`) a une complexité moyenne $O(n)$, où n est le nombre d'éléments dans la liste. Comme vous le savez certainement, les arbres binaires vus au cours ont un avantage de ce point de vue, puisque la recherche peut s'effectuer en $O(\log n)$ si l'arbre est équilibré.

Pour tirer parti de cette avantage, nous désirons écrire un ensemble de classes permettant de réaliser une structure de données associative utilisant les arbres binaires. L'idée est simplement de stocker dans chaque nœud de l'arbre une paire clef/valeur, en plus du fils gauche et du fils droit. Le sous-arbre gauche d'un arbre contient tous les nœuds dont la clef est inférieure à la clef de la racine, tandis que le sous-arbre droit contient tous ceux dont la clef est supérieure à la clef de la racine.

Par exemple, la figure 1 présente un arbre binaire équivalent à la liste associative donnée dans l'exercice précédent. Notez que l'arbre binaire présenté ici n'est pas le seul qui soit équivalent à cette liste, mais il est par contre le seul à être équilibré.

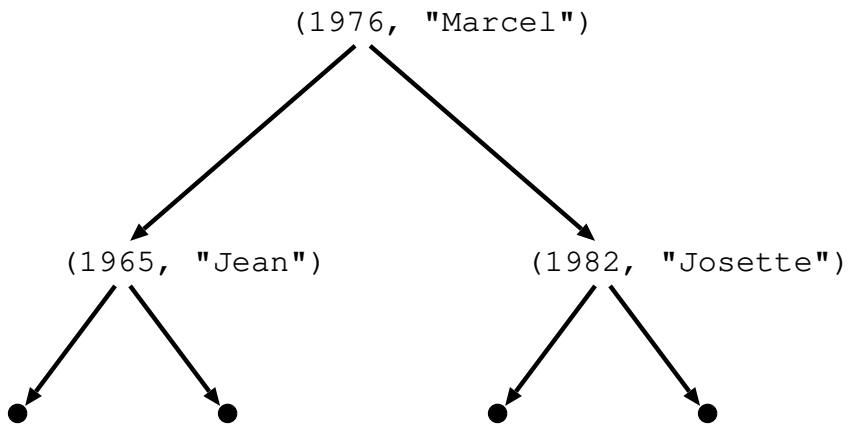

FIG. 1 – Arbre binaire

Complétez le squelette de programme ci-dessous pour terminer les méthodes `insert` et `lookup` de la classe `IntMap`. Lors d'une insertion, si la clef qu'on essaie d'ajouter est déjà dans l'arbre, la nouvelle entrée doit remplacer l'existante. Dans cet exercice, on se limitera au cas où les clefs sont des nombres entiers.

```
abstract class IntMap[A] {
  def insert(key: Int, value: A): IntMap[A] = // à compléter
  def lookup(key: Int): A = // à compléter
};

case class Empty[A]
  extends IntMap[A];

case class Node[A](left: IntMap[A],
                   keyVal: Pair[Int, A],
                   right: IntMap[A])
  extends IntMap[A];
```

Notez bien que, contrairement à la semaine passée, vous utiliserez la décomposition par filtrage de motifs plutôt que la décomposition orientée-object.