Programmation Fonctionnelle : TD4

EFREI - Paris

Ensembles ordonnés et opérations ensemblistes

*
* *

Préambule

Dans ce TD, on va ré-implémenter certains opérateurs classiques de l'algèbre ensembliste :

- L'*intersection* $E_1 \cap E_2 = \{e | e \in E_1 \wedge e \in E_2\}$, les éléments dans E_1 et E_2
- L'*union* $E_1 \cup E_2 = \{e | e \in E_1 \vee e \in E_2)$, les éléments dans E_1 ou E_2.
- La *différence* $E_1 \backslash E_2 = \{e | e \in E_1 \wedge e \notin E_2\}$, les éléments de E_1 qui ne sont pas dans aussi E_2.
- La *différence symétrique* $E_1 \triangle E_2 = \{e | e \in E_1 \vee e \in E_2 \wedge \neg(e \in E_1 \wedge e \in E_2)\}$, les éléments dans E_1 ou E_2 mais pas dans les deux.
- Le *produit cartésien* $E_1 \times E_2 = \{(x, y) | x \in E_1 \wedge y \in E_2\}$, l'ensemble des couples dont la première composante appartient à E_1 et la seconde à E_2.

Pour représenter des ensembles de nombres, on peut utiliser les listes que l'on a déjà manipulées lors du TD précédent. On rappelle que les ensembles sont des structures qui sont dépourvues de doublon. On précise également ici que l'on considère des ensembles d'entiers ***ordonnés*** !
Le fait d'établir une relation d'ordre sur les éléments de la liste est très important ici car cela va nous permettre de diminuer la complexité de nos algorithmes.
En effet pour des ensembles non ordonnés, l'algorithme trivial (c'est-à-dire non optimisé) de l'intersection serait de parcourir toutes les paires possibles d'éléments résultant des deux ensembles. On obtient dans ce cas un algorithme de complexité en $O(n \times m)$ où $|E_1| = n$ et $|E_2| = m$ sont les tailles respectives des deux ensembles. Ainsi, pour des ensembles très grands, un algorithme comme celui-ci est inefficace.
Dans le cas où ces deux ensembles sont au préalable triés, il est possible de créer des algorithmes efficaces de complexité linéaire en $O(n)$.

Exemple : On considère les ensembles d'entiers triés suivants représentés par des listes.

```
# let e1 = [0; 1; 2; 3; 5; 6; 7; 9];;
val l1 : int list = [0; 1; 2; 3; 5; 6; 7; 9]
# let e2 = [2; 3; 4; 8; 9];;
val l2 : int list = [2; 3; 4; 8; 9]
```

Problème 1

1. Vérification des conditions :

   (a) Écrire la spécification et le code d'une fonction `estTriee l` qui prend en paramètre une liste d'entiers l est qui retourne *vrai* si et seulement si l est triée selon la relation d'ordre $<$.

(b) Écrire la spécification et le code d'une fonction `estEnsemble l` qui prend en paramètre une liste d'entiers l est qui retourne *vrai* si et seulement si l est sans doublon d'élément.

(c) Écrire la spécification et le code d'une fonction `estEnsembleOrdonné l` qui prend en paramètre une liste d'entiers l est qui retourne *vrai* si et seulement si l est un ensemble ordonné.

2. Génération d'ensembles ordonnés :

(a) Écrire la spécification et le code d'une fonction `last l` qui retourne le dernier élément d'une liste l.

(b) Écrire la spécification et le code d'une fonction `alea k sup` qui tire un nombre entier aléatoire $a \in]\!]k, sup]\!]$, c'est-à-dire tel que $a > k$ et $a \leq sup$. On supposera que $k < sup$.
On utilisera la fonction `Random.int sup` qui génère un entier aléatoire $b \in [\![0, sup[\![$.

(c) Ecrire une fonction récursive qui utilise la fonction alea et qui renvoie une liste de taille n remplie avec des nombres aléatoires.

Problème 2

Écrire le code Ocaml des fonctions suivantes réalisant les opérations ensemblistes classiques, sur des ensembles ordonnés représentés par des listes. Attention, vos fonctions devront fournir elles aussi des ensembles ordonnés, sans doublon et ordonnés.

1. `intersection e1 e2;;`

```
- : int list = [2; 3; 9]
```

2. `union e1 e2;;`

```
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

3. `difference e1 e2;;`

```
- : int list = [0; 1; 5; 6; 7]
```

4. `diff_sym e1 e2;;`

```
- : int list = [0; 1; 4; 5; 6; 7; 8]
```

5. `prod_cartesien e1 e2;;`[1]

```
- : (int * int) list =
[(0, 2); (0, 3); (0, 4); (0, 8); (0, 9); (1, 2); (1, 3); (1, 4); (1, 8); (1, 9); (2,
2); (2, 3); (2, 4); (2, 8); (2, 9); (3, 2); ...; (9, 4); (9, 8); (9, 9)]
```

[1] Résultat donné dans l'ordre lexicographique.