TME3 – Représentation des ensembles finis par des listes sans doublons

Lorsque E est un ensemble fini contenant n éléments, il est possible de le représenter en extension par une liste $[e_1; \cdots; e_n]$ correspondant à une certaine énumération de ses éléments. Il existe donc plusieurs listes différentes qui représentent le même ensemble. Par exemple les listes $[1; 2]$ et $[2; 1]$ représentent toutes les deux l'ensemble $\{1, 2\}$. Un élément ne peut pas apparaître plus d'une fois dans une liste représentant un ensemble (toutes les listes ne représentent donc pas un ensemble, par exemple la liste $[1; 1; 1]$ ne correspond pas à la représentation d'un ensemble par une liste). Dans les questions qui suivent les ensembles sont représentés par des listes sans doublons.

Exercice 3.1.

1. Définir une fonction récursive de signature
   ```
   is_in (e: 'a) (l : 'a list) : bool
   ```
   qui étant donnés un élément `e` et une liste `l` représentant un ensemble détermine si `e` appartient à l'ensemble représenté par `l`.
   Exemples :
   - `(is_in 4 [])` donne `false`
   - `(is_in 3 [1;3;2])` donne `true`
   - `(is_in 4 [1;3;2])` donne `false`

   Remarque : on suppose ici que l'égalité entre deux éléments de la liste `l` peut être testée avec l'opérateur = de OCaml.
2. En utilisant la fonction `is_in`, définir une fonction de signature
   ```
   add_elem (e : 'a) (l : 'a list) : 'a list
   ```
   qui étant donnés un élément `e` et une liste `l` représentant un ensemble ajoute l'élément `e` en tête de la liste `l` s'il n'apparaît pas déjà dans `l`.
   Exemples :
   - `(add_elem 4 [1;3;2])` donne la liste `[4;1;3;2]`
   - `(add_elem 4 [1;3;4;2])` donne la liste `[1;3;4;2]`
   - `(add_elem 4 [])` donne la liste `[4]`

Exercice 3.2 (Sous-ensemble et égalité d'ensembles)**.** Etant donnés deux ensembles A et B, on dit que A est inclus dans B, ou encore que A est un sous-ensemble de B, si et seulement si tous les éléments appartenant à A appartiennent aussi à B, ce que l'on note $A \subseteq B$.

$$A \subseteq B \Leftrightarrow (\forall x \quad x \in A \Rightarrow x \in B)$$

1. Définir une fonction récursive de signature
   ```
   is_subset_rec (l1 : 'a list) (l2 : 'a list) : bool
   ```
   permettant de déterminer si l'ensemble représenté par la liste `l1` est inclus dans l'ensemble représenté par la liste `l2`.
   Exemples :
   - `(is_subset_rec [4;2] [0;2;4;6])` donne `true`
   - `(is_subset_rec [2;3;4] [0;2;4;6])` donne `false`

— `(is_subset_rec [] [0;2;4;6])` donne `true`
— `(is_subset_rec [4;2;0;6] [0;2;4;6])` donne `true`

2. Définir une fonction `is_subset` correspondant à une version non récursive de la fonction `is_subset_rec`, qui utilise les fonctions `List.for_all` et `is_in` (il s'agit de vérifier que tous les éléments de `l1` apparaissent dans `l2`).
3. Deux ensembles A et B sont égaux s'ils contiennent exactement les mêmes éléments c-à-d si $A \subseteq B$ et $B \subseteq A$. En utilisant la fonction `is_subset`, définir une fonction de signature
   `eq_set (l1: 'a list) (l2: 'a list) : bool`
   permettant de déterminer si l'ensemble représenté par la liste `l1` est égal à l'ensemble représenté par la liste `l2`.
   Exemples :
   — `(eq_set [4;2] [0;2;4;6])` donne `false`
   — `(eq_set [4;2;6;0] [0;2;4;6])` donne `true`
   — `(eq_set [4;2;6;0] [0;2;5;6])` donne `false`
   — `(eq_set [] [])` donne `true`

Exercice 3.3 (Intersection)**.** L'intersection de deux ensembles A et B, notée $A \cap B$, est l'ensemble des éléments appartenant à la fois à A et à B.

$$A \cap B = \{e \mid e \in A \text{ et } e \in B\}$$

1. Définir une fonction récursive de signature
   `intersection_rec (l1 : 'a list) (l2 : 'a list) : 'a list`
   permettant de construire une liste représentant l'intersection des ensembles représentés par les listes `l1` et `l2`.
   Exemples :
   — `(intersection_rec [1;2;3;4] [3;5;4;6])` donne `[3;4]`
   — `(intersection_rec [] [3;5;4;6])` donne `[]`
   — `(intersection_rec [1;2;3;4] [])` donne `[]`
   Remarque : l'ordre d'apparition des éléments dans la liste construite dépend de l'ordre avec lequel le code traite les éléments des listes, il peut donc être différent de celui des listes données ci-dessus.
2. Définir une fonction `intersection` correspondant à une version non récursive de la fonction `intersection_rec`, qui utilise les fonctions `List.filter` et `is_in` (il s'agit de ne conserver d'une des deux listes que les éléments apparaissant dans l'autre).

Exercice 3.4 (Union)**.** L'union de deux ensembles A et B, notée $A \cup B$, est l'ensemble des éléments appartenant à à A ou appartenant à B.

$$A \cup B = \{e \mid e \in A \text{ ou } e \in B\}$$

1. En utilisant la fonction `add_elem` (et sans utiliser la fonction `is_in`), définir une fonction récursive de signature
   `union_rec (l1 : 'a list) (l2 : 'a list) : 'a list`
   permettant de construire une liste représentant l'union des ensembles représentés par les listes `l1` et `l2`.
   Exemples :

— `(union_rec [1;2;3;4] [3;5;4;6])` donne `[1;2;3;4;5;6]`

— `(union_rec [] [3;5;4;6])` donne `[3;5;4;6]`

Remarque : l'ordre d'apparition des éléments dans la liste construite dépend de l'ordre avec lequel le code traite les éléments des listes, il peut donc être différent de celui des listes données ci-dessus.

2. Définir une fonction `union_left` correspondant à une version non récursive de la fonction `union_rec`, qui utilise les fonctions `List.fold_left` et `add_elem` (il s'agit d'ajouter sans créer de doublons à l'une deux listes tous les éléments apparaissant dans l'autre).
3. Définir une fonction `union_right` correspondant à une version non récursive de la fonction `union_rec`, qui utilise les fonctions `List.fold_right` et `add_elem`.

Exercice 3.5 (Produit cartésien)**.** Le produit cartésien de deux ensembles A et B, noté $A \times B$, est l'ensemble des couples (a, b) tels que $a \in A$ et $b \in B$.

$$A \times B = \{(a, b) \mid a \in A \text{ et } b \in B\}$$

1. En utilisant la fonction `List.map`, définir une fonction non récursive de signature
   ```
   make_pairs (x : 'a) (l : 'b list) : ('a * 'b) list
   ```
   qui construit la liste de couples $[(x, e_1); \cdots; (x, e_n)]$ lorsque `l` est la liste $[e_1; \cdots; e_n]$.
   Exemples :
   — `(make_pairs true [3;5;4;6])` donne `[(true,3);(true,5);(true,4);(true,6)]`
   — `(make_pairs 1 [])` donne `[]`
2. Définir une fonction récursive de signature
   ```
   product_rec (l1 : 'a list) (l2 : 'b list) : ('a * 'b) list
   ```
   permettant de construire une liste représentant le produit cartésien des deux ensembles A et B représentés par les listes `l1` et `l2`. Pour chaque élément $x \in A$, il suffit de construire la liste de couples $[(x, y_1); \cdots; (x, y_k)]$ où $[y_1; \cdots; y_k]$ est la liste représentant l'ensemble B, puis de concaténer les listes obtenues.
   Exemples :
   — `(product_rec [1;2;3] ['a';'b'])` donne
   `[(1, 'a'); (1, 'b'); (2, 'a'); (2, 'b'); (3, 'a'); (3, 'b')]`
   — `(product_rec [1;2;3] [])` donne `[]`
   — `(product_rec [] [1;2;3])` donne `[]`
   Remarque : l'ordre d'apparition des éléments dans la liste construite dépend de l'ordre avec lequel le code traite les éléments des listes, il peut donc être différent de celui des listes données ci-dessus.
3. Définir une fonction `product` correspondant à une version non récursive de `product_rec`, qui utilise les fonctions `List.map`, `make_pairs` et `List.flatten` (`List.flatten` permet de conacténer les listes présentes dans une liste de listes).

Exercice 3.6 (Ensemble des parties)**.** Etant donné un ensemble E, une partie A de E est un sous-ensemble de E. On note $\wp(E)$ l'ensemble des parties de E. Un élément $A \in \wp(E)$ est donc un ensemble A tel que $A \subseteq E$. Puisque pour tout ensemble E on a $\emptyset \subseteq E$ et $E \subseteq E$, $\wp(E)$ contient toujours les éléments $\emptyset$ et E (en particulier, on a $\wp(\emptyset) = \{\emptyset\}$).

$$\wp(E) = \{A | A \subseteq E\}$$

L'ensemble $\wp(E)$ peut être construit récursivement : si E est l'ensemble vide, alors $\wp(E)$ est simplement le singleton contenant l'ensemble vide, sinon, E peut être vu comme l'union d'un ensemble E_1 (éventuellement vide) avec un singleton $\{x\}$, c-à-d $E = \{x\} \cup E_1$. La propriété suivante permet d'obtenir $\wp(E)$ à partir de l'ensemble $\wp(E_1)$:

$$\wp(E) = \wp(\{x\} \cup E_1) = \wp(E_1) \cup \{\{x\} \cup A | A \in \wp(E_1)\}$$

1. Définir une fonction récursive de signature

   ```
   powerset_rec (l : 'a list) : 'a list list
   ```

   permettant de calculer la liste représentant l'ensemble des parties d'un ensemble représenté par la liste `l`.

   Exemples :

   — `(powerset_rec [])` donne `[[]]`

   — `(powerset_rec [1;2;3])` donne `[[];[3];[2];[2;3];[1];[1;3];[1;2];[1;2;3]]`

   Remarque : l'ordre d'apparition des éléments dans la liste construite dépend de l'ordre avec lequel le code traite les éléments des listes, il peut donc être différent de celui des listes données ci-dessus.

2. Définir une fonction `powerset` correspondant à une version non récursive de la fonction `powerset`, qui utilise les fonctions `List.fold_left` et `List.map`.