Feuille d'exercices n°4

Exercice 1. Tri et réduction

Soient les fonctions `inserer` et `reduction` suivantes :

```
# let rec inserer = fun ord e l ->
  match l with
  | [] -> [e]
  | h::t as l -> if (ord e h) then e::l else h::(inserer ord e t);;

# let rec reduction = fun operator neutral l ->
  match l with
  | [] -> neutral
  | h::t -> operator h (reduction operator neutral t);;
```

Écrire une fonction de tri par insertion en utilisant uniquement ces deux fonctions.

Exercice 2. Réductions

En cours, nous avons supposé que l'opération donnée en argument à `reduction` est une opération associative. Si ce n'est pas le cas, on peut avoir deux réductions possibles donnant éventuellement deux résultats différents. La bibliothèque standard d'Objective Caml offre deux fonctions :

```
List.fold_left: ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
List.fold_right: ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

Lorsque l'opération n'est pas associative, les résultats peuvent différer selon la réduction qu'on utilise :

```
# List.fold_left ( - ) 0 [1;2;3;4];;
- : int = -10
# List.fold_right ( - ) [1;2;3;4] 0;;
- : int = -2
# List.fold_left (fun x y -> "(f "^x^" "^y^")") "e" ["x0";"x1";"x2";"x3";"x4"];;
- : string = "(f (f (f (f (f e x0) x1) x2) x3) x4)"
# List.fold_right (fun x y -> "(f "^x^" "^y^")") ["x0";"x1";"x2";"x3";"x4"] "e";;
- : string = "(f x0 (f x1 (f x2 (f x3 (f x4 e)))))"
```

1. À quelle fonction correspond notre fonction `reduction` (l'ordre des arguments peut être différent) ? Selon la réponse, la renommer `reduction_gauche` ou `reduction_droite`.
2. Écrire la fonction qui manque : `reduction_gauche` ou `reduction_droite` sans utiliser les fonctions `List.fold_left` et `List.fold_right`.

Exercice 3. Écrire une fonction `cut_list l a b` qui renvoie la sous-liste des éléments de `l` à partir de l'indice `a` inclus jusqu'à l'indice `b` non compris. Donner le type de `cut_list`.

Exercice 4. Une liste d'association est une liste de type `('a *('b list)) list`, associant des objets de type `'b` à des objets de type `'a`.

Exemple :

$$\texttt{let assoc\_list} = [(\text{"}\texttt{Pierre}\text{"}, [1;2;3]); (\text{"}\texttt{Paul}\text{"}, [3;1;7]); (\text{"}\texttt{Jacques}\text{"}, [9;4;1])]$$

Remarques :

— (a) Un même élément de type `'a` ne doit pas apparaître deux fois dans la liste, par exemple

$$[(\text{"Pierre"}, [1;2;3]); (\text{"Pierre"}, [4;2;1])]$$

n'est **pas** une liste d'association.

— (b) Intuitivement, tous les éléments de type `'a` sont associés à une liste d'éléments de type `'b`, même s'ils n'apparaissent pas dans la liste. Dans cet exemple, "Michel" est associé à la liste vide.

Questions :

1. Écrire la fonction `recherche : 'a → ('a*'b list) list → 'b list` qui renvoie les éléments de type `'b` associés à un élément de type `'a` dans une liste d'association.

$$\underline{\text{exemple}} : \texttt{recherche"Paul"assoc\_list} = [3;1;7]$$

2. Écrire la fonction `change :`

$$'\texttt{a} \to ('\texttt{b list} \to' \texttt{b list}) \to ('\texttt{a} *' \texttt{b list})\ \texttt{list} \to ('\texttt{a} *' \texttt{b list})\ \texttt{list}$$

qui modifie la liste des éléments de type `'b` associés à un élément de type `'a` en appliquant une fonction de type `'b list -> 'b list` à cette liste.

$$\underline{\text{exemple}} : \quad \texttt{change"Pierre"(fun l} \to 5 :: \texttt{l)\ assoc\_list} = [(\text{"Pierre"}, [5;1;2;3]); (\text{"Paul"}, [3;1;7]); (\text{"Jacques"}, [9;4;1])]\ .$$

3. En utilisant la fonction `change`, écrire la fonction

$$\texttt{add} :' \texttt{a} \to' \texttt{b} \to ('\texttt{a} *' \texttt{b list})\ \texttt{list} \to ('\texttt{a} *' \texttt{b list})\ \texttt{list}$$

qui ajoute un élément de type `'b` à la liste des éléments associés à un élément de type `'a` (voir la remarque (b) pour le cas où l'élément de type `'a` n'apparaît pas dans la liste d'association avant l'ajout). Si l'élément de type `'b` est déjà associé à l'élément de type `'a`, aucune modification n'est faite.

$$\text{exemple :} \quad \begin{array}{l} \texttt{add "Paul" 8 assoc\_list} = \\ [(\text{"Pierre"}, [2;3;4]); (\text{"Paul"}, [8;3;1;7]); (\text{"Jacques"}, [9;4;1])] \\ \texttt{add "Paul" 7 assoc\_list} = \\ [(\text{"Pierre"}, [2;3;4]); (\text{"Paul"}, [3;1;7]); (\text{"Jacques"}, [9;4;1])] \end{array}$$

4. En utilisant la fonction `change`, écrire la fonction

$$\texttt{mapAssoc} :' \texttt{a} \to ('\texttt{b} \to' \texttt{b})\ \texttt{list} \to ('\texttt{a} *' \texttt{b list})\ \texttt{list} \to ('\texttt{a} *' \texttt{b list})\ \texttt{list}$$

qui applique une fonction de type `'b → 'b` à tous les éléments de type `'b` associés à un élément de type `'a` dans une liste d'association.

$$\text{exemple :} \quad \begin{array}{c} \texttt{mapAssoc "Pierre" (funx} \to \texttt{x} + 1)\texttt{assoc\_list} = \\ [(\text{"Pierre"}, [2;3;4]); (\text{"Paul"}, [3;1;7]); (\text{"Jacques"}, [9;4;1])] \end{array}$$