Feuille de TP n°8

```
module type DICTIONARY =
  sig
    type ('a, 'b) t

    (** Retourne un dictionnaire vide. *)
    val empty: ('a, 'b) t

    (** Teste si le dictionnaire passé en paramètre est vide. *)
    val is_empty: ('a, 'b) t -> bool

    (** (mem key dictionary) renvoie true si le dictionnaire contient
        la clé key, et false sinon. *)
    val mem: 'a -> ('a, 'b) t -> bool

    (** (find_opt key dictionary) renvoie (Some value) si la valeur
        value est associée à la clé key dans le dictionnaire, et
        renvoie None sinon. *)
    val find_opt: 'a -> ('a, 'b) t -> 'b option

    (** (find key dictionary) renvoie une valeur si cette valeur
        est associée à la clé key dans le dictionnaire, et lève
        l'exception Not_found sinon. *)
    val find: 'a -> ('a, 'b) t -> 'b

    (** (remove key dictionary) renvoie un dictionnaire dans
        lequel la clé key n'est plus associé à aucune valeur. *)
    val remove: 'a -> ('a, 'b) t -> ('a, 'b) t

    (** (add key value dictionary) ajoute une association entre
        key et value dans le dictionary. Si une association
        existait déjà avec la même clé, cette précédente association
        est supprimée. *)
    val add: 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
  end
```

FIGURE 1 – Type de module `DICTIONARY`

Exercice 1 – Module dictionnaire : listes

Écrire un module `ListAssoc` qui implante le type de module `DICTIONARY` donné à la figure 1 en ayant pour `t` la définition suivante :

```
type ('a, 'b) t = ('a * 'b) list
```

Exercice 2 – Module dictionnaire : arbres binaire de recherche

Un arbre binaire est un arbre binaire de recherche si pour tout nœud la valeur contenue à ce nœud est supérieure à toutes les valeurs contenues dans son enfant gauche, et inférieure à toutes les valeurs contenues dans son enfant droit.

Écrire un module `TreeAssoc` qui implante le type de module `DICTIONARY` donné à la figure 1 en ayant pour `t` la définition suivante :

```
type 'a binary_tree =
  | Empty
  | Node of 'a * 'a binary_tree * 'a binary_tree

type ('a, 'b) t = ('a * 'b) binary_tree
```

et dans lequel toutes les opérations fournies manipulent des arbres binaires de recherche. Notamment, après la suppression ou l'ajout d'une association dans un arbre binaire de recherche, l'arbre retourné doit être un arbre binaire de recherche. Les opérations de recherche doivent tenir compte du fait que l'arbre binaire est un arbre binaire de recherche pour être plus efficaces.

Exercice 3 – Retour sur l'évaluation des expressions

Dans la feuille précédente, les types suivants avaient été définis pour manipuler des expressions arithmétiques avec variables :

```
type unary_operation = | Opp | Abs | Sqr

type binary_operation = | Add | Sub | Mul | Div | Mod | Max

type 'a expression =
  | Var of 'a
  | Int of int
  | Unary of unary_operation * 'a expression
  | Binary of binary_operation * 'a expression * 'a expression
```

1. Écrire un *foncteur* `Evaluation` qui prend en argument un module `Env` de type `DICTIONARY` et qui fournit en particulier une fonction :

   ```
   val evaluate: 'a expr -> ('a, int) Env.t -> int
   ```

   qui évalue une expression arithmétique avec variables, et qui obtient les valeurs associées aux variables grace au dictionnaire (appelée ici environnement). Si jamais l'expression contient une variable qui n'est pas dans le dictionnaire, une exception appropriée est levée, qui indique notamment la variable qui n'est pas définie dans le dictionnaire.
2. Appliquer ce foncteur au module `TreeAssoc`, créer un environnement donnant des valeurs aux variables x, y, a et b, puis évaluer les exemples d'expressions donnés dans la feuille précédente.
3. Faire de même avec le module `ListAssoc`.

Exercice 4 – Bonus – Évaluation des expressions

Dans le foncteur de l'exercice précédent, définissez les deux fonctions suivantes :

```
(** (variables e) renvoient la liste des variables contenues
    dans l'expression e. Une variable n'apparait qu'une seule
    fois dans la liste, même si elle est présenté plusieurs
    fois dans l'expression. *)
val variables : 'a expr -> 'a list

(** (make_env liste_variables) demande à l'utilisateur de
    saisir une valeur entière pour chaque variable présente
    dans la liste (on suppose qu'il n'y a pas plusieurs
    occurences de la même variable dans la liste) et qui
    créé un environnement contenant les associations ainsi
    définies. *)
val make_env: 'a list -> ('a * int) Env.t
```

Remarque : la fonction `make_env` ne peut pas être exécutée dans `https://try.ocamlpro.com`.