Programmation fonctionnelle
TP n° 7

Exercice 1 (File)

Une file est une structure de données FIFO. On peut y insérer des éléments et les récupérer un par un dans le même ordre. On peut les représenter par l'interface :

```
module type PersistantQueue = sig
  type 'a t (* Le type des files *)
  exception Empty_queue (* Exception levée par les fonctions peek et
   dequeue en cas de file vide *)
  val empty_queue : 'a t (* La file vide *)
  val is_empty : 'a t -> bool (* Prédicat pour les files vides *)
  val peek : 'a t -> 'a (* Renvoie l'élément en tête de file *)
  val enqueue : 'a -> 'a t -> 'a t (* Enfile un élément en fin de file *)
  val dequeue : 'a t -> 'a t (* Supprime l'élément en tête de file *)
end
```

1. On se propose d'écrire un module qui implémente les files de manière fonctionnelle (sans effet de bord).

   ```
   module ListQueue : Queue = struct
     type 'a t = {front : 'a list; rear : 'a list}
     ...
   end
   ```

   Le champ `front` représente les premiers éléments de la file dans l'ordre. Le champ `rear` représente les derniers éléments de la file dans l'ordre inverse. La file contenant les élements `e1`, `e2` et `e3` (dans cet ordre) peut indistinctement être représentée par :
   — `{front = [e1;e2;e3]; rear = []}`,
   — `{front = [e1;e2]; rear = [e3]}`,
   — `{front = [e1]; rear = [e3;e2]}` ou
   — `{front = []; rear = [e3;e2;e1]}`.

   (a) Expliquez l'intérêt d'une telle représentation.

   (b) Implémentez le module `ListQueue`. N'oubliez pas que vous pouvez ajouter autant de fonctions auxiliaires que vous le souhaitez : elles seront invisibles en dehors du module.

2. On se propose de réaliser un module qui implémente les files de manière impérative. Les mises à jour se font par mutation.

   ```
   module LinkedQueue = struct
     type 'a node = {value : 'a; mutable next : 'a node option}
     type 'a t = {mutable first : 'a node option; mutable last : 'a node
      option}
     ...
   end
   ```

   L'implémentation proposée utilise un chaînage simple comme les listes classiques. Il est cependant possible d'ajouter un nouveau nœud en fin de liste. Pour ce faire, la file possède un pointeur sur son dernier

nœud (en plus du premier) et les nœuds ont leur champ `next` mutable. La file vide est représentée par `{first = None; last = None}` : elle ne contient aucun nœud. Pour une file ne contenant qu'un seul élément, le premier et le dernier nœud sont identiques **en mémoire**.

(a) Définissez une interface `MutableQueue` adaptée aux files impératives. Inspirez-vous de l'interface `PersistantQueue`.

(b) Implémentez le module `LinkedQueue`.

Exercice 2 (Interface `Comparable`)

On se donne l'interface `Comparable` qui permet de représenter un ensemble d'éléments munis d'une relation d'ordre.

```
module type Comparable = sig
  type t
  val compare : t -> t -> int
end
```

1. Définissez un foncteur `ComparableCouple` paramétré par des modules `C1` et `C2` qui implémentent l'interface `Comparable`. Le foncteur renvoie un module implémentant `Comparable` sur les couples d'éléments `C1.t * C2.t`. Il faut implémenter un ordre lexicographique : $(a, b) < (c, d) \Leftrightarrow a < c$ ou $(a = c$ et $b < d)$.
2. Définissez un foncteur `ComparableList` paramétré par un module `C` qui implémente `Comparable`. Le foncteur renvoie un module implémentant `Comparable` sur les listes d'éléments de `C.t`. Il faut implémenter un ordre lexicographique (les listes ne sont pas nécessairement de la même taille).

Exercice 3 (Dictionnaire)

Un dictionnaire est une structure de données qui permet d'associer des clés et des valeurs.

```
module type Map = sig
  type key (* Le type des clés *)
  type 'a t (* Le type des dictionnaires, les valeurs sont de types 'a *)
  val empty : 'a t (* Le dictionnaire vide *)
  val add : key -> 'a -> 'a t -> 'a t (* Ajoute ou met à jour une
    association clé/valeur au dictionnaire *)
  val find : key -> 'a t -> 'a option (* Renvoie la valeur associée à une
    clé dans le dictionnaire *)
  val remove : key -> 'a t -> 'a t (* Retire une clé du dictionnaire *)
end
```

Il existe de nombreuses implémentations différentes des dictionnaires basées sur des stratégies diverses. On se propose d'en réaliser une utilisant des arbres binaires de recherche.

Pour pouvoir naviguer dans l'arbre, les clés doivent implémenter l'interface `Comparable`. On va donc travailler sur un foncteur `TreeMap` :

```
(* with type rend publique l'instanciation du type key *)
module TreeMap (Key : Comparable) : Map with type key = Key.t = struct
  type key = Key.t
  type 'a node = {key : key; value : 'a; left : 'a t; right : 'a t}
  and 'a t = Empty | Node of 'a node
  ...
end
```

On rappelle que, pour un nœud interne `{key; value; left; right}`, toutes les clés apparaissant dans `left` (resp. `right`) sont **strictement** inférieures (resp. **strictement** supérieures) à `key`. La valeur `value` est celle associée à `key`.

En vous aidant de vos notes de TD, implémentez le foncteur `TreeMap`.