

Séance 7: MODULES ET FONCTEURS

L3 – Université Nice Sophia Antipolis

Exercice 1 (Files persistantes, **)

On considère la signature ci-dessous pour un module de files persistantes.

```

1 module type PersistantQueue = sig
2   type 'a t (* Le type des files *)
3   exception Empty_queue (* Exception levee par les fonctions peek et
4     dequeue en cas de file vide *)
5   val empty_queue : 'a t (* La file vide *)
6   val is_empty : 'a t -> bool (* Predicat pour les files vides *)
7   val peek : 'a t -> 'a (* Renvoie l'element en tete de file *)
8   val enqueue : 'a -> 'a t -> 'a t (* Enfile un element en fin de file
9     *)
10  val dequeue : 'a t -> 'a t (* Supprime l'element en tete de file *)
11 end

```

On se propose d'implémenter une file persistante par un couple de listes

```

1 type 'a t = {front : 'a list; rear : 'a list}

```

Le champs `front` représente les premiers éléments de la file dans l'ordre. Le champs `rear` représente les derniers éléments dans l'ordre inverse. La file contenant les éléments `e1`, `e2`, `e3` (dans cet ordre) peut indistinctement être représentée par

- {front = [e1;e2;e3]; rear = []}
- {front = [e1;e2]; rear = [e3]}
- {front = [e1]; rear = [e3;e2]}
- {front = []; rear = [e3;e2;e1]}

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

Exercice 2 (Interface comparable, **)

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

```

1 module type Comparable = sig
2   type t
3   val compare : t -> t -> int
4 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 $(a = c \text{ et } b < d)$.

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`. Il faut implémenter un ordre lexicographique (les listes ne sont pas nécessairement de même taille).

□

Exercice 3 (Dictionnaire, **)

On considère la signature ci-dessous pour un module de dictionnaire.

```
1 module type Map = sig
2   type key (* le type des clés *)
3   type 'a t (* le type des dictionnaires, les clés sont de type 'a *)
4   val empty : 'a t (* le dictionnaire vide *)
5   val add : key -> 'a -> 'a t -> 'a t (* ajoute ou met à jour une
6     association clé/valeur *)
7   val find : key -> 'a t -> 'a option (* renvoie la valeur associée à
8     une clé dans le dictionnaire *)
9   val remove : key -> 'a t -> 'a t
10 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` :

```
1 module TreeMap (Key:Comparable) : Map with type key = Key.t = struct
2   type key = Key.t
3   type 'a node = {key: key; value: 'a; left: 'a t; right: 'a t}
4   and 'a t = Empty | Node of 'a node
5   ...
6 end
```

On rappelle que pour un noeud `{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`.

Complétez l'implémentation du module `TreeMap`.

□