Programmation 1 – TP n° 2
Modules et Foncteurs

1 Tableaux, mémoire

1.1 Présentation

Le tableau est une structure de données de base qui existe dans beaucoup de langages de programmation. Un tableau de taille n de type A correspond intuitivement à un ensemble de n cellules numérotées de 0 à $n-1$, chacune contenant un objet de type A.

1	7	0	9	1	7	4	3	2	9	0	3	1	7	9	4	0	1	1	2	1	9	5	5

FIG. 1 – Un tableau d'entiers de taille 24.

Plusieurs opérations sont disponibles :
- créer un tableau de taille n (dans notre contexte, un tableau a une taille fixée et on ne peut pas la changer),
- récupérer l'objet en i-ème position,
- changer la valeur d'une cellule d'un tableau.

Cette dernière opération peut être comprise de différentes façons :
- soit elle change le tableau,
- soit elle renvoie un nouveau tableau.

On parlera de tableau *modifiable* dans le premier cas, et *fonctionnel* dans le second cas.

Le but de cette partie est de proposer plusieurs représentations possibles des tableaux modifiables et fonctionnels, et d'examiner comment on passe d'une structure à l'autre.

1.2 Tableaux modifiables

Un module qui permet de représenter un tableau modifiable a la signature suivante :

```
module type MTABLEAU =
sig
    type 'a tableau
    val create : int -> 'a -> 'a tableau
    val length : 'a tableau -> int
    val get : 'a tableau -> int -> 'a
    val set : 'a tableau -> int -> 'a -> unit
end
```

Détaillons les différentes fonctions :
- « `create n x` » renvoie un tableau de taille `n` dont toutes les cases contiennent la même valeur `x`.

- « `length t` » renvoie la taille du tableau `t`.
- « `get t i` » renvoie la valeur de la `i`-ème case du tableau `t`. Si `i` est une valeur non autorisée (c'est-à-dire supérieure ou égale à la taille du tableau, ou négative), la fonction lève l'exception `Out_of_bounds`. (on rappelle que les cases du tableau sont numérotées de 0 à $n-1$ où n est la longueur du tableau).
- « `set t i x` » modifie le tableau `t` de sorte que la valeur `x` soit désormais stockée à la `i`-ème case du tableau. Cette fonction modifie ses arguments et ne renvoie rien.

Voilà une utilisation typique d'une telle structure de données :

```
(* A est un module de signature MTABLEAU *)
(* Creation d'un tableau de chaines de caracteres
   de 6 cases initialisees avec la valeur "oui" *)
let t = A.create 6 "oui";;
A.length t;;     (* renvoie 6 *)
A.get t 2;;      (* renvoie "oui" *)
A.set t 4 "non";;
A.get t 4;;      (* renvoie "non" *)
let m = t;;
A.set m 4 "pourquoi";;
```

Q 1.1 Que renverrait alors `A.get t 4;;`? Expliquer pourquoi.

1.2.1 Représentation par des listes de références

Les tableaux modifiables sont déjà implémentés dans OCaml, dans le module `Array`. Nous allons les coder avec des listes de références. Ainsi le tableau | 1 | 8 | 7 | 1 | est représenté par la liste :

```
let t = [ ref(1); ref(8); ref(7); ref(1) ]
```

Q 1.2 Écrire un module nommé `MRefList` de signature `MTABLEAU` utilisant cette représentation des tableaux.

```
module MRefList : MTABLEAU =
struct
    type 'a tableau = 'a ref list
    let rec create n x = ...
    ...
end
```

Q 1.3 Vérifier sa correction avec des exemples bien choisis.

1.3 Tableaux fonctionnels

Contrairement aux tableaux modifiables, on ne peut pas modifier un tableau fonctionnel : demander à changer la i-ème cellule d'un tableau fonctionnel renvoie un nouveau tableau qui ne diffère du précédent que par cette cellule.

Un module qui permet de représenter un tel tableau a la signature suivante :

```
module type FTABLEAU =
sig
```

```
        type 'a tableau
        val create : int -> 'a -> 'a tableau
        val length : 'a tableau -> int
        val get : 'a tableau -> int -> 'a
        val set : 'a tableau -> int -> 'a -> 'a tableau
    end
```

Un exemple d'utilisation d'un tel module :

```
(* A est un module de signature FTABLEAU *)
(* Creation d'un tableau de chaines de caracteres
   de 6 cases initialisees avec la valeur "alpha" *)
let t = A.create 6 "alpha";;
A.length t;; (* renvoie 6 *)
A.get t 2;;  (* renvoie "alpha" *)
let m = A.set t 4 "omega";;
A.get t 4;;  (* renvoie "alpha" *)
A.get m 4;;  (* renvoie "omega" *)
```

1.3.1 Fonctions

La façon la plus naturelle de représenter un tableau fonctionnel est d'utiliser les listes ; comme c'est trop facile, on ne le fera pas dans ce TP… Une des manières les plus élégantes[1] de représenter un tableau est de le représenter par une fonction de type `int -> 'a`. Ainsi le tableau | 1 | 0 | 8 | 8 | peut être représenté par l'une des deux fonctions suivantes :

```
let f n = if n >= 0 and n < 2 then (1-n)
          else if n >= 0 and n < 4 then 8
          else raise Out_of_bounds

let g n = if n >= 0 and n < 4 then List.nth [1;0;8;8] n
          else raise Out_of_bounds
```

Q 1.4 Écrire et vérifier sur des exemples un module nommé `FFun` de signature `FTABLEAU` utilisant cette représentation des tableaux. (Commencez par écrire la fonction `length`, qui n'est pas la plus simple.)

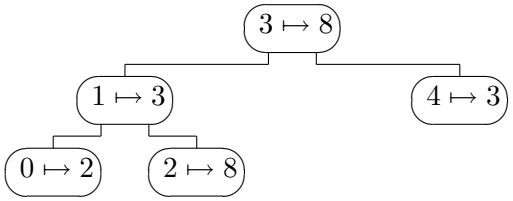
1.3.2 Arbres binaires de recherche

Il s'agit sans doute de la structure la plus utilisée pour représenter ces tableaux. La structure est un arbre binaire de couples `(i,x)` (signifiant que l'élément en position i est l'élément `x`) vérifiant la contrainte suivante : En tout nœud `(i,x)`, le fils gauche ne contient que des éléments `(j,y)` qui ont une position ($j < i$) et le fils droit ne contient que des éléments `(j,y)` qui ont une position ($j > i$). On définit donc le type comme suit :

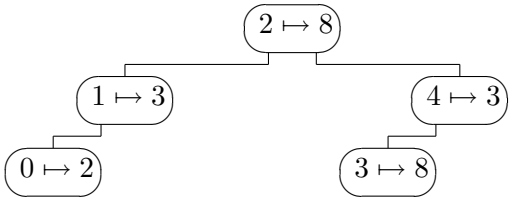
```
type 'a tableau =
  | Leaf
  | Node of int * 'a * 'a tableau * 'a tableau
```

[1] Mais aussi une des moins efficaces

Voici deux exemples d'arbres binaires de recherche représentant le même tableau :




```
let t = Node(3,8,
          Node(1,3 ,
            Node(0,2, Leaf, Leaf),
            Node(2,8, Leaf, Leaf)),
          Node(4,3 , Leaf, Leaf))
```




```
let t = Node(2,8,
          Node(1,3,
            Node((0,2), Leaf, Leaf),
            Leaf),
          Node(4,3,
            Node(3,8, Leaf, Leaf),
            Leaf))
```

Q 1.5 Écrire et vérifier un module nommé `FArbre` de signature `FTABLEAU` utilisant cette représentation des tableaux.
Remarque. Il y a plusieurs façons de faire `create`. Essayez d'écrire la meilleure…

1.4 Foncteurs

On va maintenant construire des foncteurs permettant de passer d'un des modèles à l'autre.

Q 1.6 Écrivez un foncteur `MtoF` qui convertit un module de type `MTABLEAU` en un module de type `FTABLEAU` (faites simple, cf. la suite) :

```
module MtoF (M : MTABLEAU) : FTABLEAU =
struct
   ...
end
```

Q 1.7 Écrivez un foncteur `FtoM` qui convertit un module de type `FTABLEAU` en un module de type `MTABLEAU`. Que dire de la composition des deux foncteurs ?

2 Avec des ensembles ordonnés

Les tableaux précédents sont polymorphes et peuvent donc contenir des valeurs de types arbitraires. Les modules (et les foncteurs) permettent d'ajouter de la structure à ces types. Ainsi le type des 'types comparables' peut s'écrire :

```
module type CompType =
sig
    type t
    val compare : t -> t -> bool
end
```

Q 2.1 Ecrire un module `TABLEAU` dépendant de `CompType` permettant en plus de trier la structure.

3 Des monades

Les monades sont un type de données abstrait permettant d'ajouter de la structure aux types. Elles servent par exemple à décrire des théories comme les monoïdes, les groupes, les fonctions partielles, les exécutions d'un programme, etc. Elles sont usuellement représentées dans les langages de programmation fonctionnelle par la signature suivante :

```
module type MonadF =
sig
    type 'a t
    val bind : 'a t -> ('a -> 'b t) -> 'b t
    val return : 'a -> 'a t
end
```

Moralement, le type `'a t` est le type `'a` enrichi avec une certaine structure ; la fonction `return` permet d'encapsuler une valeur dans cette structure ; la fonction `bind` permet d'agir sur les valeurs à l'intérieur de cette structure. De plus, une monade doit vérifier que :

- `return` préserve les informations de son argument, *i.e.*
  $\forall m : \texttt{'a t},\ \texttt{bind}\ m\ \texttt{return} = m$ et
  $\forall x : \texttt{'a},\ \forall f : \texttt{'a} \to \texttt{'b t},\ \texttt{bind}\ (\texttt{return}\ x)\ f = fx$ et que
- `bind` se comporte bien avec la composition de fonctions, *i.e.*
  $\forall m : \texttt{'a t},\ \forall f : \texttt{'a} \to \texttt{'b t},\ \forall g : \texttt{'b} \to \texttt{'c t},$
  $\texttt{bind}\ (\texttt{bind}\ m\ f)\ g = \texttt{bind}\ m\ (\texttt{fun x} \to \texttt{bind}\ (f\ x)\ g).$

Une autre présentation des monades, venue de l'algèbre, est décrite par la signature :

```
module type MonadA =
sig
    type 'a t
    val return : 'a -> 'a t
    val join : 'a t t -> 'a t
    val map : ('a -> 'b) -> 'a t -> 'b t
end
```

Les équations que doivent vérifier cette structure sont :

$\texttt{map id} = \texttt{id}$
$\forall f : \texttt{'a} \to \texttt{'b},\ \forall g : \texttt{'b} \to \texttt{'c}, (\texttt{map}\ g) \circ (\texttt{map}\ f) = \texttt{map}\ (g \circ f)$
$\forall m : \texttt{'a t},\ \texttt{join}\ (\texttt{return}\ m) = \texttt{join}\ (\texttt{map return}\ m) = m,$
$\forall u : \texttt{'a t t t},\ \texttt{join}\ (\texttt{join}\ u) = \texttt{join}\ (\texttt{map join}\ u).$

Le type `'a option` est défini par `None | Some of 'a`. On considère les fonctions de type `'a -> 'b option` comme des fonctions partielles de `'a` dans `'b`.

Q 3.1 Comment composer ces fonctions partielles ?

Ces opérations apparaissent naturellement dans les structures de monades, dont le type des listes et le type option font partie.

Q 3.2 Ecrire un module `OPTION : MonadF` tel que le type `'a t` soit égal à `'a option`

Q 3.3 Ecrire un module `LIST : MonadA` tel que le type `'a t` soit égal à `'a list`

Le type `'a t`, dans les signatures `Monad`, est opaque. Une fonction attendant un argument de type `'a LIST.t` ne pourra pas prendre en argument une valeur de type `'a list`. On peut expliciter les types d'un modules en utilisant les contraintes de types :

```
module LIST : MonadA with type 'a t = 'a list = struct
    ...
end
```

Q 3.4 Vérifier que ces monades sont correctes, *i.e.* vérifient les axiomes donnés précédemment. Si la réponse est non, reculez de 2 cases.

Q 3.5 Programmer deux foncteurs entres ces modules.

Q 3.6 Conservent-ils les axiomes ?

Q 3.7 Que valent leurs compositions ?