TP #17 — Quelques exercices sur les `'a array`s

Exercice 1. `Array.init`

1. Écrivez une fonction OCaml **non récursive** :
   `init : int -> (int -> 'a) -> 'a array`
   de mêmes spécifications que la fonction `Array.init`.
   Autrement dit, `init n f` s'évalue en un `'a array` nouvellement créé de longueur $n \geq 0$, et initialise l'élément d'indice $i \in [\![0, n-1]\!]$ à `f i` (de sorte que le résultat de l'évaluation est égal à `[|f 0; f 1; ...; f (n - 1)|]`.

Exercice 2. *Conversions* `list` & `array`

1. Écrivez une fonction OCaml **non récursive** :
   `to_list : 'a array -> 'a list`
   telle que `to_list a` s'évalue en une `list` nouvellement créée de mêmes éléments que a (avec même multiplicité ; dans le même ordre (c'est à dire que la tête de liste est égale à l'élément de a d'indice `0`, etc.)).
2. Écrivez une fonction OCaml **récursive & n'utilisant aucune référence** :
   `to_list' : 'a array -> 'a list`
   de mêmes spécifications que `to_list`.
3. Écrivez une fonction OCaml :
   `of_list : 'a list -> 'a array`
   telle que `of_list l` s'évalue en un un `'a array` nouvellement créé de mêmes éléments que son argument (avec même multiplicité ; dans le même ordre).
   Pour traiter (avec le bon type) le cas particulier d'une liste vide, vous pouvez utiliser le littéral « `[||]` » d'un tableau vide.
4. Écrivez une fonction OCaml :
   ```
   alist2amat : int -> (int * int list) list
                -> bool array array
   ```
   telle que pour $gS \geq 0$, gA une liste de au plus gS couples de la forme (i, ℓ_i), avec $i \in [\![0, gS - 1]\!]$ et ℓ_i des listes d'éléments $\in [\![0, gS - 1]\!]$, `alist2amat gS gA` s'évalue en un `bool array array` gM de dimensions $gS \times gS$ nouvellement créé où `gM.(i).(j) = true` ssi. gA contient un élément (i, ℓ_i) tel que ℓ_i contient un élément j.
   On peut supposer (même si cela ne change pas grand chose) que gA est construite de la façon la plus « économe » possible : pour tout i, gA ne contient qu'un élément de la forme (i, ℓ_i) et il n'y a pas de collisions dans ℓ_i.

Exemple. Pour :

```
gA = [(0, [1; 2]); (1, [0; 2]); (2, [1]) ]
```

on a `alist2amat 4 gA` qui s'évalue en :

```
[|[|false;  true;  true; false|];
  [|true;  false;  true; false|];
  [|false;  true; false; false|];
  [|false; false; false; false|]|]
```

Conseil. Vous pouvez éventuellement utiliser (plusieurs fois) l'itérateur sur les listes `List.iter` (à connaître).

5. Écrivez une fonction :

   ```
   amat2alist : bool array array
                -> int * (int * int list) list
   ```

   qui effectue la « conversion inverse » de `alist2amat`. Autrement dit, soit g_S, g_A le résultat de l'évaluation de `amat2alist gM`, on doit avoir `alist2amat gS gA` s'évaluant en un résultat identique à g_M.
   Dans la limite du raisonnable, les résultats construits par `amat2alist` doivent encore une fois être les plus « économes » possibles.

Exercice 3. *Moyenne glissante*

1. Écrivez une fonction :

   ```
   mova : float array -> int -> float array
   ```

   telle que `mova t ws` s'évalue en *tc*, un `float array` nouvellement créé égal à *t* dans le cas où $ws \leq 1$, et sinon tel que `tc.(i)` est égal à la moyenne des au plus *ws* éléments de *t* d'indices non négatifs $\in [\![i - ws + 1, i]\!]$.

Exercice 4. *~~All the~~ iterators*

1. Implémentez tous les itérateurs sur les `Array` qui sont au programme : `mem`, `exists`, `for_all`, `map`, `iter`.
2. Écrivez également une fonction OCaml :

   ```
   iteri : (int -> 'a -> unit) -> 'a array -> unit
   ```

   semblable à `iter`, si ce n'est que la fonction itérée reçoit également en premier argument l'indice de l'élément auquel elle est appliquée.

Exercice 5. *Tableaux persistants*

Cet exercice est basé sur le chapitre 4.4 de Learn Programming with OCaml.
Son but est d'implémenter une structure de donnée abstraite de *tableau persistant*. Une structure de donnée *persistante* se caractérise par le fait qu'elle est *observationnellement immuable* : comme une structure de donnée immuable (ou fonctionnelle) les opérations de « modification » renvoient une nouvelle version

de la structure (où la modification a été prise en compte), mais une structure de donnée persistante peut utiliser la mutabilité de façon « interne ».
La structure de tableau persistant est donc similaire à celle de tableau en ce qu'elle permet d'accéder et modifier des éléments d'une collection en fonction de leur indice, mais elle ajoute la fonctionnalité qu'il est possible d'accéder à une version précédente du tableau : c'est à dire qu'après une opération `set` effectuée sur un certain tableau `pa`, il sera toujours possible d'accéder au contenu de `pa` tel qu'il était avant le `set`. Ceci pourrait être réalisé en copiant intégralement le tableau à chaque `set`, c'est à dire d'implémenter cette fonction comme :

```
let set a i v = let na = copy a in na.(i) <- v ; na
```

avec `copy` une fonction de copie de profondeur satisfaisante. Cette approche a l'avantage que l'accès aux versions « historiques » est aussi efficace que pour les tableaux usuels, mais elle a un inconvénient de taille : le coût (en temps **et** en mémoire) de `set` est linéaire en la taille de `a` ; nous allons voir comment faire mieux, au prix d'accès plus coûteux pour les versions historiques.
L'idée principale de l'implémentation est la suivante :

— La version la plus récente du tableau (la plus récemment accédée, ou bien en lecture ou bien en écriture) est stockée comme un tableau mutable (par exemple un `'a array`).
— Toute autre version est stockée comme une chaîne de modifications à appliquer à la version la plus récente.

Par exemple, deux versions `pa0` et `pa1` d'un même tableau persistant correspondant respectivement aux données `[|0; 1; 2|]` et `[|1; 1; 2|]` pourraient être stockées comme un(e référence vers un) `int array` `a` contenant `[|0; 1; 2|]` pour `p0`, et une référence vers `a` et le fait que l'élément d'indice `0` doit être modifié à `1` pour `p1`.
Nous allons utiliser les types mutuellement récursifs suivants pour représenter les tableaux persistants :

```
type 'a parray = 'a data ref
and  'a data =
     | Arr of 'a array
     | Diff of int * 'a * 'a parray
```

L'utilisation de la récursion mutuelle n'est ici pas nécessaire, mais permet à l'inféreur de type d'automatiquement typer d'avantage de fonctions comme opérant sur des `'a parray` plutôt que sur des `'a data ref`.
Avec ce type, les deux tableaux persistants `p0` et `p1` de l'exemple ci-dessus pourraient être représentés de la façon suivante :

— `p0` est représenté directement comme, et utilise donc le constructeur `Arr` : `{contents = Arr [|0; 1; 2|]}`
— `p1` est représenté indirectement, et utilise donc le constructeur `Diff` pour à la fois stocker la référence vers `p0` et les changements qu'il est nécessaire

de lui appliquer :
`{contents = Diff (0, 1, p0)}`

1. Écrivez une fonction OCaml :
   `init : int -> (int -> 'a) -> 'a parray`
   telle que `init n f` s'évalue en un `'a parray` nouvellement construit, représenté explicitement par le `'a array [|f 0; ...; f (n - 1)|]` de `n` éléments.

Les fonctions `get` et `set` vont toutes deux être basées sur une fonction :
`reroot : 'a parray -> 'a array`
telle que lorsque `pa` est une référence vers un `Arr a`, `reroot pa` s'évalue simplement en `a`, et sinon « remonte l'historique » des `Diff` jusqu'à trouver un tableau stocké explicitement comme un `'a array`, le modifie afin qu'il représente explicitement le contenu référencé par `pa`, modifie tous les tableaux persistants rencontrés en remontant l'historique pour exprimer leur contenu comme des `Diffs` vers `pa`, et s'évalue finalement en le `'a array` référencé par `pa`.
On illustre le fonctionnement de cette fonction en reprenant les exemples `p0` et `p1` ci-dessus, et en leur adjoignant un tableau persistant `p2` obtenu en modifiant `p0` en son indice `1` à la valeur `0` (le contenu de `p2` quand stocké explicitement doit donc être égal à `[|0; 0; 2|]`). On a alors :

— après appel à `reroot p0` :
  — `p0 = {contents = Arr [|0; 1; 2|]}`
  — `p1 = {contents = Diff (0, 1, p0)}`
  — `p2 = {contents = Diff (1, 0, p0)}`

— après appel à `reroot p1` :
  — `p0 = {contents = Diff (0, 0, p1)}`
  — `p1 = {contents = Arr [|1; 1; 2|]}`
  — `p2 = {contents = Diff (1, 0, p0)}`

— après appel à `reroot p2` :
  — `p0 = {contents = Diff (1, 1, p2)}`
  — `p1 = {contents = Diff (0, 1, p1)}`
  — `p2 = {contents = Arr [|0; 0; 2|]}`

2. Écrivez une telle fonction `reroot`
3. Testez.
4. Écrivez une fonction OCaml :
   `get : 'a parray -> int -> a`
   telle que `get pa i` s'évalue en la valeur de l'élément d'indice `i` de `pa` (quand il existe). Cette fonction devra avoir comme effet de bord qu'à l'issue de son appel, `pa` est représenté explicitement (c'est à dire, par une référence vers un `Arr`).

5. Quel est le coût pire cas de `get` (à exprimer en fonction d'une quantité pertinente) ? Lors de deux appels successifs à `get` sur un même tableau `pa`, quel est le coût pire cas du second appel ?
6. Écrivez une fonction OCaml :

   ```
   set : 'a parray -> int -> 'a -> 'a parray
   ```

   telle que `set pa i v` s'évalue en un tableau persistant nouvellement construit obtenu en modifiant l'élément d'indice `i` de `pa` en `v`. Ce nouveau tableau devra être représenté explicitement (dans le même sens que ci-dessus).
7. Testez.
8. Quel est le coût pire cas de `set` ? Lors d'une suite d'appels :

   ```
   let pa'  = set pa  i'  v'  in
   let pa'' = set pa' i'' v'' in
   ...
   ```

   quel est le coût pire cas du second appel à `set` ?
9. Écrivez une fonction OCaml :

   ```
   length : 'a parray -> int
   ```

   qui `reroot` son argument et s'évalue en sa longueur.
10. Écrivez une fonction OCaml :

    ```
    iter : ('a -> unit) -> 'a parray -> unit
    ```

    similaire à `Array.iter`, mais opérant sur des `'a parray`