TP1 - Ordre supérieur

(*UPS - L3 Informatique - UE TAPFA*)

Remarques :

- Un réflexe (en programmation fonctionnelle typée) qui peut faciliter l'implémentation de vos fonctions : écrivez d'abord leur type…
- N'hésitez pas à solliciter votre encadrant de TP en cas de doute, ou si vous avez l'impression de rester bloqué(e) longtemps sur une question.
- Les deux fonctions a priori plus difficiles sont `trans` et `unite`.

Les matrices vues comme des listes

On considère une matrice `m` de dimension `(n, p)` où `n` est le nombre de lignes et `p` le nombre de colonnes. Par définition, c'est l'ensemble des coefficients `m_ij` pour `1 ≤ i ≤ n` et `1 ≤ j ≤ p`.

`m` peut être représentée en OCaml par une liste de listes de la forme :

```
[[m_11 ; ... ; x_1j ; ... ; x_1p];
 ... ;
 [m_i1 ; ... ; x_ij ; ... ; x_ip];
 ... ;
 [m_n1 ; ... ; x_nj ; ... ; x_np]]
```

À noter les cas particuliers de `[]` qui représente les matrices de dimension `(0, p)` et de `[];...; []` (`n`fois) qui représente la matrice de dimension `(n, 0)`.

On rappelle que la transposée de `m` est la matrice `a` de dimension `(p, n)` dont les coefficients sont définis par `a_ij = m_ji` pour tous `1 ≤ i ≤ p` et `1 ≤ j ≤ n`.

Contrainte : sauf mention du contraire, n'utilisez pas la bibliothèque standard des listes (`List.`)

0) Échauffement

Définissez les fonction `hd` (pour "head", i.e., le premier élément de la liste), resp. `tl` (pour "tail", i.e. la liste privée de son premier élément) permettant d'extraire l'élément de tête d'une liste, resp. la queue d'une liste, en levant une exception en cas d'erreur.

1) Transposition d'une matrice

a) Écrire la fonction récursive `tete` qui renvoie la liste contenant les éléments de la première colonne d'une matrice.

b) Écrire la fonction récursive `reste` qui prend en argument une matrice `m` de dimension `(n, p)` et renvoie cette matrice amputée de sa première colonne, donc de dimension `(n, p-1)`.

c) Écrire la fonction récursive `trans` qui retourne la transposée d'une matrice.

Indication si vous hésitez comment déterminer le cas d'arrêt de la fonction `trans`*:* Ouvrez le *toplevel*, définissez une matrice `m` de dimension `(2, 2)`, et essayez d'appliquer la fonction `reste` de façon répétée (d'abord, `reste m`, puis `reste (reste m)`, etc.)

2) Fonctionnelle map

a) Déduire de l'écriture de `tete` et de `reste` la fonctionnelle `map` (vue en TD) qui, étant donné une fonction unaire et une liste, construit la liste des résultats de l'application de la fonction à tous les éléments de la liste. Donner son type (entre commentaires) et ré-écrire les fonctions `tete2` et `reste2` (sans utiliser `let rec`) avec `map`.

En principe, vos fonctions `tete2` *et* `reste2` *seront bien plus concises que* `tete` *et* `reste`*…*

b) Écrire la fonction récursive `ligzero` qui étant donné n, construit la liste `[0;...; 0]` contenant n zéros.

c) Écrire, en utilisant `map` et `ligzero`, la fonction `zero` qui, étant donné n, construit une matrice *carrée* de dimension `(n, n)` dont tous les éléments sont à 0 (zéro).

d) Écrire la fonction récursive `unite` qui, étant donné n, construit une matrice unité de dimension n, c'est-à-dire, une matrice carrée de dimension n dont la diagonale est à `1` (un) et les autres éléments à zéro. On pourra utiliser `ligzero` et `map`.

Indication : comment construire la matrice `unite n` à partir de `unite (n - 1)` ?

3) Addition de deux matrices

a) Écrire une fonctionnelle récursive `map2` qui, étant données une fonction binaire et deux listes de même longueur, construit la liste des résultats de l'application de la fonction à tous les éléments de même rang des deux listes (si les deux listes sont de longueurs différentes, on lèvera une exception avec `failwith`). Donner son type (entre commentaires).

b) Écrire, en utilisant `map2`, la fonction `somlig` qui calcule la somme de 2 lignes.

c) Écrire, en utilisant `map2`, la fonction `add` qui calcule la somme de deux matrices.

4) Produit de deux matrices

a) Écrire la fonction récursive `prodligcol : int list -> int list -> int` qui calcule le produit d'une ligne avec une colonne.

b) Écrire, en utilisant `map`, la fonction `prodligtmat` qui calcule le produit d'une ligne avec une matrice donnée sous forme d'une liste de colonnes (en d'autres termes, on suppose qu'avant d'appeler cette fonction, la matrice a déjà été transposée).

c) Écrire, en utilisant `map`, la fonction `prod` qui calcule le produit de deux matrices.

5) Bonus

a) Écrire une fonction `create`, qui étant donné une fonction `f` et un entier `n`, construit la liste de taille `n` dont le ième élément (numéroté à partir de 1) est l'application de la fonction `f` à `i` :

```
create : (int -> 'a) -> int -> 'a list

create (fun n -> n) 3 = [1; 2; 3]
```

b) Écrire une fonction `couples`, qui étant donné un entier `n`, construit la matrice `C` de dimension `(n, n)` (on dit aussi "matrice carrée d'ordre `n`") définie par les coefficients `c_ij = (i, j)`, donc en particulier :

```
couples 2 = [[(1, 1); (1, 2)];
             [(2, 1); (2, 2)]]
```

c) (Ré)écrire la fonction `zero` (nommée `zero2`) en utilisant `create`.

d) (Ré)écrire la fonction `unite` (nommée `unite2`) en utilisant `create`.

6) En guise de conclusion

Ce TP était l'illustration de l'utilité de deux fonctionnelles classiques sur les listes (`List.map` et `List.map2` dans la bibliothèque standard). Leur utilisation permet une implémentation plus concise - et plus facile à lire si l'on est familier avec celles-ci.

Nous verrons dans la suite du cours d'autres fonctionnelles importantes, qui permettent des *design patterns* similaires.