épreuve d'informatique | **module 3I008** | **Université Pierre et Marie Curie**
Durée 2 heures | **MPIL** | **Année 2016-2017**

Partiel du 27 février 2017

Documents autorisés - Engins électroniques interdits

Exercice 1 : Jeu de la vie

Le jeu de la vie est un automate cellulaire visant à simuler l'évolution d'une population de cellules dans leur environnement à partir d'un ensemble réduit de règles.

- Le monde est un tore (chambre à air) c'est-à-dire que chaque case a toujours 8 voisins (diagonales comprises). Les voisins de droite de la dernière colonne sont les cases de la 1ère colonne.
- Une règle d'apparition : si une case vide a exactement 3 voisins
- Une règle de disparition : si une case pleine a moins de 2 ou plus de 3 voisins.

Seule l'ancienne génération doit être prise en compte pour le calcul de la nouvelle génération.

```
-------------------------------------------------
| I |   |   |   |   |   |   |   |   |   | Y |
-------------------------------------------------
| I |   |   |   |   |   |   |   | I |   | I |
-------------------------------------------------
|   |   |   |   | I |   | I |   |   |   |   |
-------------------------------------------------
|   |   |   |   |   | X |   |   |   |   |   |
-------------------------------------------------
|   |   |   |   |   | I |   |   | I |   |   |
-------------------------------------------------
|   |   |   |   |   |   |   |   |   |   |   |
-------------------------------------------------
|   |   |   |   |   |   |   |   |   |   | I |
-------------------------------------------------
```

La cellule `X` possède 3 cellules dans son voisinage (3x3).
La cellule `Y` possède 4 cellules dans son voisinage (3x3).

On se donne les types suivants pour les cellules et les générations :

```
type cellule = P | V
type generation = {l : int ; h : int ; p : cell array array}
```

Une cellule peut être pleine (`P`) ou vide (`V`). Une génération est un enregistrement avec trois champs correspondant à la largeur et la hauteur du monde ainsi que la matrice des cellules.

1. Ecrire une fonction `nb_voisins` qui prend une génération et les coordonnées d'une cellule, et retourne le nombre de voisins.
2. Ecrire une fonction `next_gen` qui prend une génération et retourne la génération suivante en tenant compte des règles d'évolution du jeu de la vie.

Exercice 2 : Calculs par étapes avec ressources

On cherche à implanter un modèle d'exécution de calculs par étapes, permettant ainsi de suivre l'avancement d'un calcul et des ressources consommées. Un calcul de type `'a -> 'b` pourra être dans trois états : en cours (`Int`), fini (`Final`) ou ayant échoué (`Echec`). Pour cela on définit le type suivant :

```
type ('a,'b) etat_calc = Echec | Int of 'a * 'b  | Final of 'b
```

Un calcul sera lui défini par une fonction d'étape, l'état du calcul et le coût d'une étape. On se donne pour cela le type suivant :

```
type ('a,'b) calc = { f : ('a,'b) calc -> unit ; mutable etat : ('a,'b) etat_calc; cout : int}
```

Par exemple la fonction d'étape pour factorielle peut s'écrire de la manière suivante et permet de définir un calcul avec cette fonction d'étapes :

```
  let fact_etape c =
    match c.etat with
       Int(n,r) -> if n = 1 then c.etat <- Final(r) else c.etat <- Int(n-1,n*r)
    | _ -> ()
;;
let fact_calc n = {f = fact_etape ; etat = Int(n,1); cout = 1}
;;
```

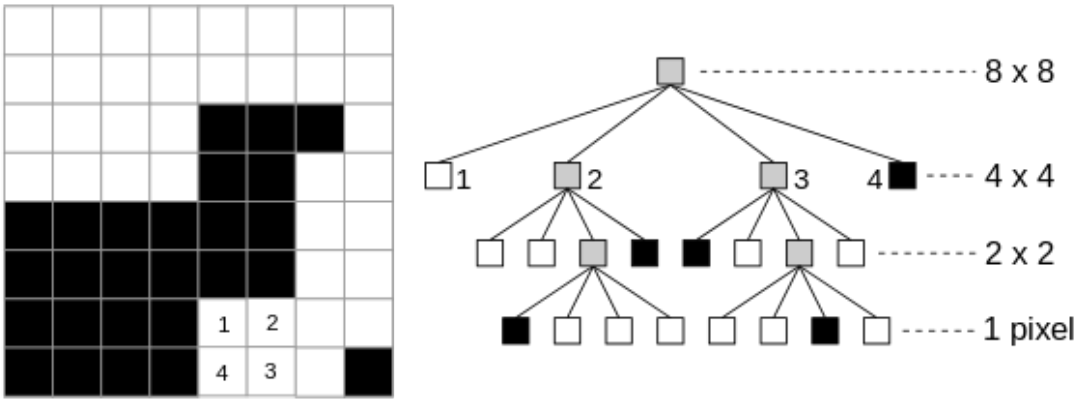
Dans toutes les questions, vous indiquerez le type des fonctions que vous écrirez.

1. Ecrire la fonction `run` qui prend une valeur de type `calc` et l'évalue par étapes jusqu'à retourner le résultat final. Si à un moment l'état du calcul passe à `Echec`, une exception est alors déclenchée. Pour cette question on ne tiendra pas compte du coût.
2. Ecrire une fonction d'étapes pour la fonction `mem` (fonction qui retourne `true` si un élément appartient à une liste et `false` sinon).
3. Construire le calcul (valeur de type `calc`) qui cherche à vérifier si le nombre `3` appartient à la liste `[1 ; 2 ; 4]`, avec un coût de 1 par étape. Tracer ensuite l'évaluation de ce calcul par la fonction `run` en indiquant les différents champs de la structure `calc` utilisée.
4. On cherche maintenant à évaluer plusieurs calculs en même temps en effectuant une étape par calcul présent dans une liste de calculs. Ecrire une nouvelle fonction `run2` qui prend une liste de calculs de même type, et qui effectue une étape de calcul par tour pour tous les calculs présents, et itère ensuite ce processus jusqu'au moment où tous les calculs ont fini ou échouent.
5. On cherche maintenant à tenir compte d'un volume de ressources. A chaque étape d'un calcul le volume de ressource diminue de son coût. Ecrire une nouvelle fonction `run3` qui prend une liste de calculs et un volume de ressources, et avance dans chaque calcul de manière équitable par rapport à son coût (un calcul dont une étape consomme 2 sera avancé deux fois moins qu'un calcul qui consomme 1). Cette fonction s'arrête quand tous les calculs sont finis ou quand le volume de ressources est consommé.

Exercice 3 : Arbres quaternaires

Un arbre quaternaire (ou quadtree) est un arbre où chaque nœud a 4 fils. On s'en sert pour coder des plans, et tout particulièrement les images. Pour cela on divise le plan en quatre quadrants égaux, puis chaque quadrant en quatre sous-quadrants et ainsi de suite ; les feuilles décrivent alors la donnée spécifique. Bien sûr quand les feuilles d'un même quadrant représentent la même information on peut regrouper cette information. Ainsi un arbre quaternaire de hauteur n peut coder une image $2^n * 2^n$, où les feuilles peuvent représenter soit une cellule d'une certaine valeur, soit une région uniforme de même valeur.

Par exemple la figure suivante [1] montre la représentation d'une image 8x8 par un arbre quaternaire :



Dans cet exercice on cherchera à implanter une bibliothèque de manipulation d'arbres quaternaires. Pour cela on se donne le type suivant pour les arbres quaternaires :

1. tirée de `https://fr.wikipedia.org/wiki/Quadtree`

```
type 'a quadrant = {no : 'a arbreQ ; so : 'a arbreQ ; ne : 'a arbreQ ; se : 'a arbreQ }
and  'a arbreQ =  Q of 'a quadrant | F of 'a ;;
```

où une feuille F(c) représente une (ou plusieurs) cellule(s) de valeur c, et où N(q) représente les quatre sous-images d'un quadrant (les labels correspondent à la position des quadrants selon les points cardinaux (nord-ouest, sud-est, ...).

1. On se donne la matrice de cellules suivantes :

```
type cell = P | V ;;

let p0 = [|
  [| V ; V ; P ; P |];
  [| V ; V ; P ; P |];
  [| P ; P;  P;  P |];
  [| V;  P;  P;  P |]
|] ;;
```

   Construire la valeur de l'arbre quaternaire équivalent.
2. Ecrire une fonction `get_cell` qui prend les coordonnées d'une cellule, une longueur et un arbre quaternaire et retourne le contenu de la cellule visée.
3. Ecrire une fonction `to_matrix` qui prend un entier (longueur), un arbre quaternaire et construit la matrice de cellules de taille $longueur * longueur$ correspondante.
4. Ecrire la fonction `same_v` qui va vérifier si l'ensemble des cellules d'une partie d'une matrice (d'un quadrant) ont toutes la même valeur. Pour cela on supposera que la longueur de la matrice carrée est une puissance de 2. Cette fonction prend une matrice de cellules, la position du coin haut-gauche (nord-ouest) et la longueur et va vérifier que l'ensemble des cases de (a,b) à (a+longueur, b+longueur) ont toutes la même valeur. Si oui elle retourne true et false sinon.
5. Ecrire une fonction `local_to_tree` qui prend un quadrant de la matrice carrée de cellules et retourne l'arbre quaternaire correspondant. Cette fonction prend en entrée la matrice, les coordonnées de coin nord-ouest et la longueur du carré à traiter, et retourne l'arbre quaternaire correspondant.
6. Ecrire une fonction `to_tree` qui prend en entrée une matrice de cellules de taille $longueur = 2^n$ et construit l'arbre quaternaire correspondant.
7. Indiquer les différents appels de fonctions lors de l'appel à `to_tree p0`.
8. En supposant que toutes les déclarations effectuées à cet exercice sont écrites dans le fichier `quadtree.ml`, donner la signature que vous désirez exposer dans le fichier `quadtree.mli`.
9. Est-ce que les arbres quaternaires construits ont bien un partage maximal (c'est-à-dire alloue au minimum) ? Si c'est le cas justifiez le, et sinon proposez des améliorations dans votre code.

Exercice 4 : Fusion des 3 exercices (Bonus)

Cet exercice n'a du sens que si vous avez effectué les trois précédents.

On cherche maintenant à utiliser des arbres quaternaires pour un jeu de la vie où une étape de travail correspond au calcul d'une nouvelle génération sachant que le coût d'une telle étape est en fonction de la taille du monde.

1. Indiquer comment feriez-vous en utilisant des arbres quaternaires du module de l'exercice 3 pour représenter le monde d'un automate cellulaire de l'exercice 1. Comparer la solution de l'exercice 1 et celle que vous proposez.
2. Ecrire une fonction (au moins la signature) pour le calcul d'une nouvelle génération d'un automate donné en tant qu'étape de calcul de l'exercice 2.
3. Ecrire la fonction de construction d'un tel calcul, où le coût est en fonction de la taille du monde.