Jeu de la vie

Introduction au Jeu de la Vie
`http://fr.wikipedia.org/wiki/Jeu_de_la_vie`

Pour cette première version, les définitions suivantes sont données (pour simplifier les modifications ultérieures) :

```
let new_cell = 1 ;;  (* alive cell *)
let empty = 0 ;;
let is_alive cell = cell <> empty ;;
```

1 Boite à outils

Listes de listes

Nous travaillons ici avec des listes de listes (appelées *matrices* ou *plateaux (boards)*).

Vous aurez besoin des fonctions suivantes (du TP 3) :

— `init_board (`l`, `c`)` *val* qui retourne une *matrice* de taille $l \times c$ remplie de *val*.
```
val init_board : int * int -> 'a -> 'a list list = <fun>
```

— `get_cell (`x`, `y`)` *board* qui retourne la valeur en position `(`x`, `y`)` dans la matrice *board*.
```
val get_cell : int * int -> 'a list list -> 'a = <fun>
```

— `put_cell` *val* `(`x`, `y`)` *board* qui remplace la valeur en `(`x`, `y`)` dans la matrice *board* par la valeur *val*.
Si la case `(`x`, `y`)` n'existe pas, *board* est retournée inchangée (pas d'exception).
```
val put_cell : 'a -> int * int -> 'a list list -> 'a list list = <fun>
```

Chargement de fichiers

Charger un fichier

Pour charger les fonctions issues des TP précédents plus facilement, utiliser la directive *#use* dans le toplevel :

```
#use "file-name";;
Read, compile and execute source phrases from the given file. This is textual inclusion: phrases
are processed just as if they were typed on standard input. The reading of the file
stops at the first error encountered.
```

Par exemple, `#use "list_tools.ml";;` chargera toutes les définitions de la section 1 du TP3 dans votre environnement CAML.

Charger un fichier en tant que module

Pour charger un fichier en tant que module, utiliser la directive *#mod_use* dans le toplevel :

```
#mod_use "file-name";;
Similar to #use but also wrap the code into a top-level module of the same name as
capitalized file name without extensions, following semantics of the compiler.
```

Par exemple, `#use "list_tools.ml";;` chargera toutes les définitions de la section 1 du TP3 dans votre environnement CAML en créant le module List_tools. Pour faire appel aux définitions du nouveau module List_tools, utiliser `List_tools.definition;;` où *definition* est le nom de la définition (que ce soit une fonction, un entier, etc ...).

Fonctions graphiques

Rappels : Tout d'abord, il faut charger le module (à ne faire qu'une seule fois) et ouvrir la fenêtre de sortie :

```
#load "graphics.cma" ;;      (* Load the library *)
open Graphics ;;             (* Open the module *)
```

`open_graph` : On peut donner en paramètres les dimensions de la fenêtre de sortie (une chaîne de caractères). La fonction suivante permet d'ouvrir une fenêtre de dimensions $size \times size$:

```
let open_window size = open_graph (" " ^ string_of_int size ^ "x" ^ string_of_int (size+20)) ;;
```

Quelques fonctions utiles (extraits du manuel[1]) :

— `val clear_graph : unit -> unit`
Erase the graphics window.

— `val rgb : int -> int -> int -> color`
`rgb` r g b returns the integer encoding the color with red component r, green component g, and blue component b. r, g and b are in the range `0..255`.

Exemple : `let grey = rgb 127 127 127 ;;`

— `val set_color : color -> unit`
Set the current drawing color.

— `val draw_rect : int -> int -> int -> int -> unit`
`draw_rect` x y w h draws the rectangle with lower left corner at x, y, width w and height h. The current point is unchanged. Raise `Invalid_argument` if w or h is negative.

— `val fill_rect : int -> int -> int -> int -> unit`
`fill_rect x y w h` fills the rectangle with lower left corner at x,y, width w and height h, with the current color. Raise Invalid_argument if w or h is negative.

De la matrice à l'affichage

Le "plateau" de jeu est une matrice $size \times size$ qui sera affichée sur la fenêtre graphique : il faut faire la correspondance entre les coordonnées dans la matrice et les coordonnées sur la fenêtre graphique.

Note : La taille du plateau sera passée en paramètre à certaines fonctions, pour éviter de la recalculer à chaque fois !

1. Écrire la fonction `draw_cell (`x`, `y`) `*size color*` qui dessine une cellule à partir de sa position (x, y) dans la matrice, sa taille (en pixels) et sa couleur *color* : un carré de coté *size* entouré de gris. Il est conseillé d'ajouter (1, 1) à (x, y) pour ne pas "coller" au cadre.

   ```
   val draw_cell : int * int -> int -> Graphics.color -> unit = <fun>
   ```

2. Écrire la fonction `draw_board` qui prend en paramètre la matrice représentant le plateau de jeu, la taille (en pixels) des cellules, et dessine le plateau sur la fenêtre graphique (penser à effacer la fenêtre...).

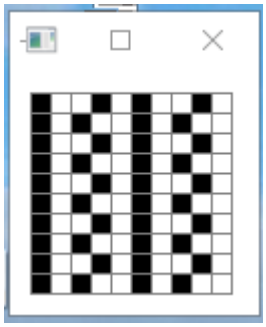
   ```
   val draw_board : int list list -> int -> unit = <fun>
   ```

   Utiliser la définition suivante :

   ```
   let cell_color = function
     | 0 -> white          (* predefined colors in Graphics *)
     | _ -> black ;;
   ```

1. https://caml.inria.fr/pub/docs/manual-ocaml-4.05/libref/Graphics.html

Exemples :



```
# let board = [[1; 1; 1; 1; 1; 1; 1; 1; 1; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
               [1; 0; 1; 0; 1; 0; 1; 0; 1; 0];
               [0; 1; 0; 1; 0; 1; 0; 1; 0; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
               [1; 1; 1; 1; 1; 1; 1; 1; 1; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0];
               [1; 0; 1; 0; 1; 0; 1; 0; 1; 0];
               [0; 1; 0; 1; 0; 1; 0; 1; 0; 1];
               [0; 0; 0; 0; 0; 0; 0; 0; 0; 0]
              ] ;;
 val board : int list list = ...

# let test_display board cell_size =
    open_window (length board * cell_size + 40) ;
    draw_board board cell_size ;;
 val test_display : int list list -> int -> unit = <fun>

# test_display board 10 ;;
 - : unit = ()
```

2 Le jeu

Les règles

À chaque étape (génération), l'évolution d'une cellule est entièrement déterminée par l'état de ses huit voisines de la façon suivante :

— Une cellule morte possédant exactement trois voisines vivantes devient vivante (elle naît).

— Une cellule vivante possédant deux ou trois voisines vivantes le reste, sinon elle meurt (elle disparaît).

1. Écrire la fonction `rules0` qui à partir d'une cellule et de son nombre de voisines retourne son nouvel état.

   ```
   val rules0 : cell:int -> near:int -> int = <fun>
   ```

2. Écrire la fonction `count_neighbours` (*x*, *y*) *board size* qui retourne le nombre de cellules vivantes (utiliser `is_alive`) autour de la cellule en (*x*, *y*) dans *board* de taille (*size*, *size*).

   ```
   val count_neighbours : int * int -> int list list -> int -> int = <fun>
   ```

   Bonus : Écrire cette fonction sans utiliser `get_cell`.

La vie

1. Écrire la fonction `seed_life` *board size nb_cell* qui place aléatoirement (utiliser la fonction `Random.int`) *nb_cell* nouvelles cellules dans le plateau *board* de taille $size \times size$.

   ```
   val seed_life : board:int list list -> size:int -> nb_cell:int -> int list list = <fun>
   ```

2. Écrire la fonction `new_board` *size nb* qui crée un nouveau plateau de jeu de taille $size \times size$ avec *nb* cellules vivantes.

   ```
   val new_board : size:int -> nb_cell:int -> int list list = <fun>
   ```

3. Écrire la fonction `next_generation` qui à partir du plateau et de sa taille applique les règles du jeu de la vie à toutes les cellules et retourne le nouveau plateau.

   ```
   val next_generation : board:int list list -> size:int -> int list list = <fun>
   ```

4. Écrire la fonction `game` *board size n* qui applique les règles du jeu de la vie sur *n* générations au plateau *board* de taille $size \times size$ et dessine le plateau à chaque génération.

   ```
   val game : board:int list list -> size: int -> n:int -> unit = <fun>
   ```

   Utiliser la définition suivante :

   ```
   let cell_size = 10 ;;     (* cell size in pixels *)
   ```

5. Écrire enfin la fonction `new_game` qui crée un nouveau jeu à partir de la taille du plateau, du nombre de cellules initiales et du nombre de générations.

   ```
   val new_game : size:int -> nb_cell:int -> n:int -> unit = <fun>
   ```

3 Bonus

Entrée/Sortie

Voici deux exemples de fonctions CAML :

```
# let write filename list =
        let oc = open_out filename in
        let rec aux = function
            [] -> close_out oc
            |e::l -> Printf.fprintf oc "%s " e; aux l
        in aux list;;
```

```
# let load name =
        let ic = open_in name in
        let try_read () =
        try Some (input_line ic) with End_of_file -> None in
            let rec loop () = match try_read () with
                Some s -> s::(loop ())
                |None -> close_in ic; []
        in loop ();;
```

La fonction `write` crée le fichier `filename` et le remplit avec tous les éléments de la liste `list` (éléments séparés par un espace). La fonction `load` charge le fichier `filename` dans une liste de string (un string par ligne).

Quelques précisions (pour plus d'informations, le manuel est votre ami !) :
— `open_out` permet d'ouvrir un fichier en écriture sous la forme d'un flux
— `open_in` permet d'ouvrir un fichier en écriture sous la forme d'un flux
— `close_out` permet de fermer un flux d'écriture
— `close_in` permet de fermer un flux de lecture
— `fprintf` permet d'écrire dans un flux à l'aide de formats particuliers
— `input_line` permet de récupérer la ligne courante du flux courant

1. Écrire la fonction `load_board` qui prend en paramètre le nom d'un fichier (dans le répertoire courant) contenant la matrice représentant le plateau de jeu et le charge dans une matrice.
2. Écrire la fonction `save_board` qui prend en paramètre le nom du fichier à créer et la matrice représentant le plateau de jeu, et qui écrit cette matrice dans le fichier. (Vous pouvez utiliser la fonction load puis convertir la liste de string obtenue ...)

Quelques ajouts

Tant qu'il y a de la vie...

Plutôt que de donner le nombre de générations en paramètres, on peut laisser le jeu tourner tant qu'il reste des cellules vivantes.

— Écrire la fonction `remaining` qui teste s'il reste des cellules vivantes dans un plateau donné.

— Modifier la fonction `new_game` : si le nombre de générations passé est 0, le jeu tournera tant qu'il restera des cellules. (Risque de récursion infinie !)

Patterns

Il existe des "schémas" connus (le clown, le canon à planeurs). On peut les "charger" à partir d'une liste de coordonnées (voir exemples en ligne).

— Écrire une fonction `init_pattern pattern size` qui crée un nouveau plateau de jeu de taille `size` à partir de la liste des coordonnées des cellules (`pattern`).

— Écrire la fonction `new_game_pattern` qui lance le jeu avec un "pattern" donné : en paramètre le plateau de jeu, sa taille et le nombre de générations.

Optimisations

1. Réécrire les dernières fonctions en évitant de redessiner la plateau à chaque génération.
2. `count_neighbours` : écrire cette fonction sans utiliser `get_cell` (elle ne doit faire qu'un parcours de la matrice).

Choix et compilation

Utilisez les fonctions d'entrées sorties (`read_int`, `print_...`) pour écrire une version compilée qui laisse le choix entre les différentes versions du jeu.

Voir un exemple en ligne.

Le manuel en ligne risque de vous être utile !

Fichier éxécutable

Pour produire un fichier éxécutable à partir de votre code CAML, utiliser la commande `ocamlc` :

```
ocamlc -c file.ml
```

Cette commande crée le code objet file.cmo. A partir de ce code objet il est possible de créer un code éxécutable en utilisant

```
ocamlc file.cmo -o file.o
```

où `file` est le nom du code éxécutable. Pour éxécuter `file.o`, utiliser la commande `ocamlrun` :

```
ocamlrun file.o
```

Pour compiler plusieurs fichiers, utiliser ocamlc en passant les fichiers .cmo par ordre de dépendance. Par exemple si le fichier file2.ml utilise des fonctions de file1.ml, il faudra effectuer successivement les commandes suivantes :

```
ocamlc -c file1.ml
ocamlc -c file2.ml
ocamlc file1.cmo file2.cmo -o file.o
```