Programmation Fonctionnelle
TP 7 – Morpion, partie impérative

Introduction

L'objectif de ce TP est de terminer l'implémentation du morpion. Plus précisément, nous allons implémenter :

- le module de matrice,
- l'affichage du morpion, à l'aide du module `Graphics` d'Ocaml,
- la gestion de la souris et du clavier afin de pouvoir interagir avec.

Les fonctions nécessaires du module `Graphics` seront rappelées dans le fil de l'énoncé, au moment où on en aura besoin. Il faudra utiliser les fichiers que vous aviez écrit dans la première partie de ce TP, et qui implémentent les règles du jeu, soit du morpion simple, soit du méta-morpion. Commencez donc par télécharger les fichiers en question (soit pour le morpion simple seulement, soit pour le morpion simple *et* le méta-morpion) et placez-les dans le dossier obtenu par extraction de l'archive (avec les autres fichiers `.ml`).

Si vous utilisez macOS sur votre propre machine, il est nécessaire d'installer `XQuartz` pour pouvoir utiliser le module `Graphics`.

Fonctions pratiques

Le module `Graphics` se fait vieux, et arbore donc une interface parfois peu pratique... c'est pourquoi nous vous fournissons les fonctions utilitaires suivantes.

```
(* Dessine la chaine s aux coordonnées (x,y) *)
draw_string_at x y s

(* Crée une nouvelle fenêtre de largeur w et de hauteur h *)
create_window w h

(* Laisse la fenêtre ouverte, jusqu'à ce qu'une touche du clavier
   ou un bouton de la souris soit pressé *)
close_after_event ()
```

La documentation du module `Graphics` est disponible à l'adresse :
https://ocaml.github.io/graphics/graphics/Graphics/.

Question 0 : matrices

Avant de commencer la partie graphisme du morpion, nous allons implémenter les matrices, qui avaient été utilisées dans la première partie du TP. Compléter le fichier `matrix.ml` en implémentant les fonctions suivantes :

```
type 'a t = 'a array array

(* Créée une matrice carrée de taille n,
   dont l'élément à la position (x,y) est f (x,y) *)
let init n f = failwith "TODO"

(* Renvoie l'élément de la matrice à la position (x,y) *)
let get matrice (x,y) = failwith "TODO"

(* Renvoie la taille de la matrice *)
let size matrice = failwith "TODO"

(* Change la valeur de l'élément de matrice à la position (x,y) en v *)
let set matrice (x,y) v = failwith "TODO"
```

Indication : `Array.make_matrix dimx dimy e` crée un tableau de tableaux de taille `dimx` par `dimy`, et dont toutes les cases contiennent `e`. La documentation du module `Array` est à cette adresse : https://caml.inria.fr/pub/docs/manual-ocaml/libref/Array.html. La valeur à une position (i, j) dans une matrice `m` est celle de l'expression `m.(i).(j)`. L'écriture d'une valeur `v` à cette position est une action de type `unit`, qui s'écrit `m.(i).(v) <- v`.

Question 1 : cases

À partir de maintenant, vous travaillerez dans le fichier `morpion.ml` et vous lancerez l'exécution du code avec la commande `./morpion.sh`.

1. Commençons par afficher une simple case du morpion, qui peut être remplie ou non. Chaque case est un carré de 50 pixels de côté. Écrire une fonction

   ```
   affiche_case : int option -> (int * int) -> unit
   ```

   telle que `affiche_case contenu (x,y)` affiche une case à la position `(x,y)`. Si `contenu` est `None`, la case doit être vide, et si `contenu` est `Some p`, où `p` est l'entier désignant le joueur qui a joué dans cette case, alors la case doit contenir ce nombre. Il faut utiliser `draw_rect x y w h` du module `Graphics`, qui affiche un rectangle dont le coin inférieur gauche est `(x,y)`, de largeur `w` et de hauteur `h`, et la fonction `draw_string_at` qu'on vous donne.
2. Tester cette fonction à l'aide du code suivant :

   ```
   let main () =
     create_window 800 800;
     clear_graph ();
     affiche_case (Some 1) (300,300);
     synchronize ();
     close_after_event ()

   let _ = main ()
   ```

La fonction `clear_graph` efface la fenêtre, la remplissant de blanc, et la fonction `synchronize` rend visible toutes les opérations de dessin effectuées après l'appel à `clear_graph ()`.

3. Modifier la fonction `affiche_case` pour qu'elle prenne un paramètre supplémentaire indiquant si cette case est celle où le dernier coup a été joué, et qui affiche le contenu de cette case en rouge si c'est le cas. Il faut utiliser la fonction `set_color` avec `black` et `red`. Le type de la fonction devient :

   ```
   affiche_case : int option -> bool -> (int * int) -> unit
   ```

Question 2 : plateau

Écrire une fonction :

```
affiche_morpion_dim_1
   :  int option Matrix.t
   -> (int * int) option
   -> (int * int)
   -> unit
```

qui affiche la grille de morpion représentée par la matrice passée en paramètre. Le deuxième paramètre est la position du dernier coup joué s'il existe, et le dernier paramètre est la position où il faut commencer à dessiner le plateau (le coin inférieur gauche). Il faut bien sûr utiliser la fonction `affiche_case`.

Pour tester cette fonction, on pourra modifier la fonction main et utiliser la matrice de test suivante :

```
let test_matrix =
  Matrix.init 3 (fun (i,j) ->
    if (i+j) mod 3 = 0 then None else Some ((i+j) mod 2)
  )
```

Pour que cela soit plus joli, et plus clair lorsqu'on utilisera cette fonction pour dessiner les méta-morpions, dessiner le bord avec une ligne plus épaisse, comme sur la capture d'écran en figure 1. Pour changer l'épaisseur du trait, il faut utiliser la fonction `set_line_width: int -> unit`.

Question 3 : interactivité

On sait afficher un plateau de morpion. Il nous reste à pouvoir y jouer.

1. Écrire une fonction

   ```
   plateau_apres_coup : plateau -> coup -> unit
   ```

   qui modifie le plateau pour prendre en compte un nouveau coup. On supposera que le coup en question est *légal*, au sens de la fonction `coup_legal` de la première partie du TP.

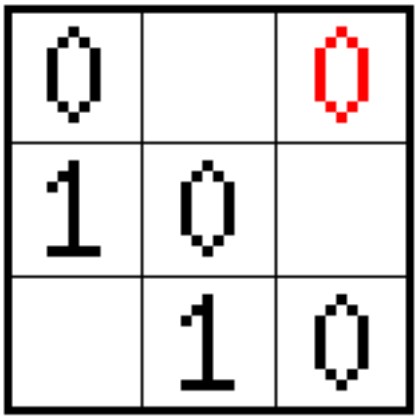

FIGURE 1 – Exemple de morpion

2. Si le plateau est affiché à partir des coordonnées `(50, 50)`, alors la case où a cliqué le joueur est donnée par :

```
let e = wait_next_event [Button_down] in
let x = (e.mouse_x - 50) / 50
and y = (e.mouse_y - 50) / 50 in ...
```

Modifier la fonction `main` pour qu'elle implémente un jeu de morpion (simple) complet. Nous vous conseillons de le faire de la manière suivante :

```
let main () =
  create_window 800 800;

  (* initialisation *)

  while (* le jeu n'est pas terminé *) do
    clear_graph ();
    (* afficher le morpion *)
    synchronize ();
    (* déterminer où le joueur a cliqué *)
    (* mettre à jour le plateau étant donné le coup joué *)
  done;
  clear_graph ();
  (* afficher le morpion *)
  synchronize ();
  close_after_event ()
```

Indication 1 : la section `(* initialisation *)` doit déclarer une variable pour le plateau qu'il faudra initialiser à l'aide de `plateau_initial` et une référence pour le dernier coup (vous pouvez aussi stocker la taille du plateau et le nombre de joueurs dans deux variables supplémentaires).
Indication 2 : il faudra utiliser les fonctions définies dans la première partie, en l'occurrence `coup_legal`, `coup_des_coordonnees_absolues`, `termine`.

3. Afficher pourquoi la partie est terminée (match-nul / vainqueur) avant l'appel à `close_after_event ()`.

Question 4 : méta-morpion

Copiez votre fichier `morpion.ml` dans un nouveau fichier que vous nommerez `metamorpion.ml` et que vous pourrez exécuter avec la commande `./metamorpion.ml`.

Adapter les deux questions précédentes au cas du méta-morpion. Les étapes principales sont les suivantes :

1. Ecrire une fonction `affiche_morpion_dim_2` qui affiche un plateau de méta-morpion. Bien sûr, cette fonction utilisera `affiche_morpion_dim_1` pour afficher ses sous-morpions.
2. Adapter la fonction `plateau_apres_coup` et la fonction `main`, notamment en utilisant les variantes `_dim2` des fonctions de la première partie du TP.
3. *(Bonus)* faire en sorte que les sous-morpions où le prochain coup peut être joué ait un fond bleu clair par exemple. Pour dessiner un rectangle plein, il faut utiliser la fonction `fill_rect` du module `Graphics`. Pour déclarer une nouvelle couleur, on peut écrire :

   ```
   let bleu_clair = rgb 164 230 235
   ```

   Indication : penser à utiliser la fonction `sous_morpion_valide`.

Question 5 *(Bonus)* : Undo/Redo

Le but de cette dernière question est d'implémenter un système d'*undo/redo*. En pratique, si l'utilisateur presse la combinaison `Ctrl-Z`, le dernier coup joué sera annulé, et s'il presse `Ctrl-Y`, le dernier coup *annulé* sera rejoué (sauf si un joueur a rejoué entre temps).

1. Tout d'abord, cela implique qu'il faut être capable de lire les entrées clavier. Il faut pour cela changer les paramètres qu'on donne à la fonction `wait_next_event` :

   ```
   let e = wait_next_event [Button_down ; Key_pressed] in
   ```

   Ensuite, il faut regarder si une touche a été pressée, et déterminer laquelle :

   ```
   if e.keypressed then (
     if int_of_char e.key = 26 then
       (* ctrl-z a été pressé *)
     else if int_of_char e.key = 25 then
       (* ctrl-y a été pressé *)
   )
   ```

   Modifier votre fonction `main` pour qu'un message soit écrit dans la console si l'une des deux combinaisons est pressée.
2. Écrire une fonction `plateau_avant_coup` qui modifie le plateau passé en argument pour annuler l'action de la fonction `plateau_apres_coup`.

3. Dans la fonction `main`, remplacer la référence stockant le dernier coup par deux références : l'une contenant la liste des coups qui ont été joués (et qui n'ont pas été annulés), et l'autre qui contient la liste des coups annulés. Notons que si des coups sont annulés et qu'ensuite un coup est joué, il faut effacer la liste des coups annulés. On se servira des fonctions `plateau_avant_coup` et `plateau_apres_coup` pour mettre à jour le plateau lorsqu'un coup est annulé / rejoué.