Programmation fonctionnelle
TP n° 8 : Le démineur

Le démineur est un jeu vidéo de réflexion popularisé par les versions 3.1 à XP de Microsoft Windows. Il se joue sur une grille rectangulaire où chaque case peut soit contenir une mine soit être vide. Le joueur gagne s'il parvient à localiser toutes les mines sans en faire exploser.

À chaque tour, le joueur sélectionne une case. Si elle contient une mine, il perd. Si elle n'en contient pas, le nombre de mines adjacentes (horizontalement, verticalement et en diagonal) est indiqué. De plus, si ce nombre est nul (aucune mine n'est proche), les cases adjacentes sont révélées de manière récursives. Si le nombre de cases non révélées est égale au nombre de mines recherchées, le joueur gagne.

Le but de ce TP est de donner une implémentation basique de ce jeu.

Partie 1 : Représentation des cases

On représente une case par un objet de la classe `case`.

```
class virtual case repr =
  object (self)
    (* Indique si la case est cachée ou a été révélée. *)
    val mutable hidden = true
   (* Accesseur du champ hidden. *)
    method is_hidden () = hidden
    (* La représentation courante de la case (un caractère). *)
    method get_repr () = if self#is_hidden () then ' ' else repr
    (* La manière dont la case réagit lorsqu'on la révèle. Le premier
       booléen renvoyé indique si une mine a explosé.  Le deuxième indique
       si les cases adjacentes doivent être récursivement révélées. *)
    method virtual interact : unit -> (bool * bool)
  end
```

Le mot-clé **`virtual`** est nouveau et n'a pas été présenté en cours. Une classe virtuelle ne peut pas être instanciée (**`new`** est interdit) et peut contenir des méthodes virtuelles. Une méthode virtuelle n'a pas de corps, seule sa signature est donnée. Les classes qui héritent d'une classe virtuelle doivent être elles-mêmes virtuelles ou implémenter toutes les méthodes virtuelles de leur classe mère. Pour ce TP, il suffit de se dire que la classe virtuelle `case` impose un contrat que ses classes filles devront respecter.

1. Définissez une classe `mine` qui représente une case contenant une mine. La représentation d'une mine est un astérisque.
2. Définissez une classe `empty` qui représente une case vide. Les cases vides sont paramétrées par un chiffre qui indique le nombre de mines adjacentes (entre 0 et 8). Ce nombre sert également de représentation.

Partie 2 : Fonctions utilitaires

La surface de jeu est représentée par une matrice de cases. Nous allons définir un certain nombre de fonctions utilitaires.

1. Définissez une fonction `get_dimension` telle que `get_dimension m` renvoie la dimension de la matrice passée en paramètre sous la forme d'un couple.
2. Définissez une fonction `is_inside` telle que `is_inside m (i, j)` renvoie `true` si `(i, j)` sont des coordonnées valides pour la matrice `m`.
3. Définissez une fonction `get_neighbors` telle que `get_neighbors m (i, j)` renvoie la liste des coordonnées adjacentes à celles passées en paramètre :
   — Dans un des coins de la matrice, une case a 3 voisins.
   — Sur un bord, elle a 5 voisins.
   — A l'intérieur, elle a 8 voisins.
4. Définissez une fonction `random_coords` telle que `random_coords m` renvoie des coordonnées aléatoires valides de la matrice `m`.

Partie 3 : Génération aléatoire d'une partie

Les paramètres d'une partie sont la taille de la surface de jeu et le nombre de mines présentes. Les mines sont placées aléatoirement dans des cases vides. Pour générer une nouvelle partie, on passe par une représentation intermédiaire de la surface de jeu : une matrice d'entiers. Un élément aux coordonnées (i, j) vaut -1 si une mine est présente, sinon l'entier correspond au nombre de mines adjacentes.

La matrice est initialement vide, elle ne contient que des zéros. On place les mines une à une en incrémentant de un toutes les cases adjacentes qui ne contiennent pas de mine. Une fois le processus fini, on crée la matrice de cases correspondantes.

1. Définissez une fonction `incr_tab` telle que `incr_tab m (i, j)` incrémente la valeur de `m` aux coordonnées `(i, j)` si elle est différente de `-1`.
2. Définissez une fonction `random_mine` telle que `random_mine m` ajoute une mine à un emplacement libre de la matrice `m`.
3. Définissez une fonction `init_case` telle que `init_case k` renvoie une case correspondant à l'entier `k`.
4. Définissez une fonction `init_game` telle que `init_game w h n` renvoie une matrice de cases de dimension `(w, h)` contenant `n` mines.

Partie 4 : La classe principale

La classe `minesweeper` représente l'état du jeu à un moment donné.

```
class minesweeper w h n =
  object (self)
    (* Le nombre de cases encore à révéler pour gagner. *)
    val mutable counter = w * h - n
    (* Indique si la partie est finie. *)
    val mutable endgame = false
    (* La surface de jeu. *)
    val cases = init_game w h n
end
```

1. Définissez une méthode `display` qui affiche l'état courant de la surface de jeu. Un exemple d'affichage est donné Figure 1.
2. Définissez une méthode `interact` telle que `interact (i, j)` met à jour l'état du jeu :
   — Si l'utilisateur interagit avec une mine, il perd la partie et on affiche un message.
   — S'il interagit avec une case vide, elle est révélée. Ses voisines le sont récursivement jusqu'à être adjacentes à une mine.
   — Le compteur de cases à révéler est décrémenté du nombre de cases révélées. S'il atteint 0, l'utilisateur a gagné, on affiche un message et la partie prend fin.

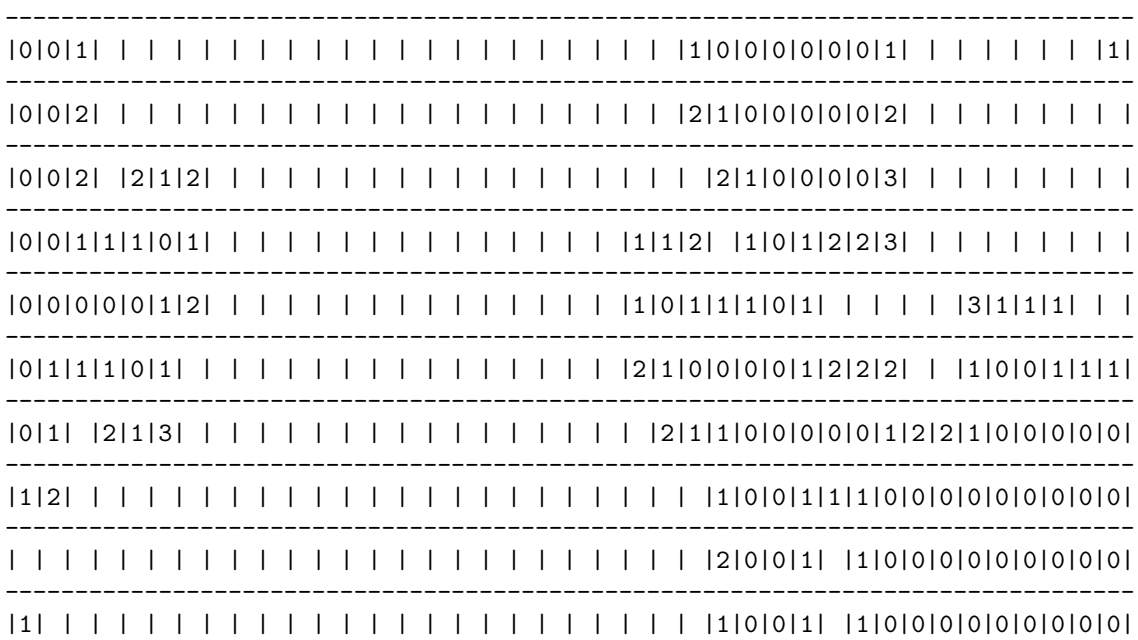

FIGURE 1 – Un exemple d'affichage pour un jeu de taille (40, 10) contenant 50 mines. L'utilisateur a déjà interagi avec chacun des coins.

Vous avez désormais la base de la logique du démineur. Vous êtes libres d'ajouter des fonctionnalités si vous le souhaitez (interface conviviale, marqueurs pour les cases, génération au premier coup pour qu'il ne soit pas un échec ...).