TP 4 : labyrinthe (à faire sur 2 séances)

Ce TP est à déposer à la fin de cette séance sur Moodle. Vous pouvez également déposer une version améliorée de votre TP jusqu'à la fin de cette semaine.

Soit un jeu de labyrinthe dans lequel un labyrinthe est représenté par une matrice. Une matrice représente un **labyrinthe valide** si elle possède les caractéristiques suivantes.

- Les seules valeurs possibles pour les cases de la matrice sont `Libre`, `Mur` et `Sortie`.
- La valeur `Sortie` représente la case de sortie du labyrinthe. Il y a une sortie unique dans le labyrinthe.
- La valeur `Mur` représente une case occupée par un mur.
- La case `Libre` représente une case libre, dans laquelle le joueur peut se déplacer.

La position d'un joueur dans le labyrinthe est représentée par deux entiers i et j, tels que i désigne la ligne de la matrice et j désigne la colonne de la matrice. La **position** d'un joueur est **valide** si i et j sont des indices valides de la matrice, si le labyrinthe est valide et si la position n'est pas celle d'un mur.

Les 4 déplacements possibles d'un joueur depuis une case sont les suivants : en haut, en bas, à droite, à gauche.

Il est conseillé d'utiliser le prédicat `valid_index` vu au TP3 plutôt qu'écrire la formule correspondante.

Partie I : spécification

0. Récupérer le fichier à compléter `tp4.mlw`. Définir un type `case` représentant une case du labyrinthe, un type `laby` représentant un labyrinthe, ainsi qu'un type `direction` définissant les déplacements possibles d'un joueur.
1. Définir le prédicat `laby_valide` indiquant si une matrice est un labyrinthe valide.

   ```
   predicate laby_valide (b : laby) = (* à compléter *)
   ```

2. Définir le prédicat `position_valide` indiquant si la position d'un joueur dans le labyrinthe est valide.

   ```
   predicate position_valide (b : laby) (i j : int) = (* à compléter *)
   ```

3. Définir le prédicat `sortie_trouvée` indiquant si le joueur se trouve à la sortie.

   ```
   predicate sortie_trouvée (b : laby) (i j : int) = (* à compléter *)
   ```

4. En utilisant le prédicat `sortie_trouvée`, spécifier le sous-programme `gagne` indiquant si le joueur se trouve à la sortie (et a donc gagné la partie).

   ```
   val gagne (b : laby) (i j : int) : bool   (* à compléter *)
   ```

5. Spécifier le sous-programme `peut_se_deplacer_vers` renvoyant un booléen indiquant si le joueur peut se déplacer dans le labyrinthe vers la case de coordonnées i et j (i.e., s'il ne se trouve pas sur un mur).

   ```
   val peut_se_deplacer_vers (b : laby) (i j : int) : bool
   ```

6. Spécifier le sous-programme `deplacement` renvoyant une position du joueur, étant données une position initiale du joueur et une direction. Si le déplacement du joueur est possible, le sous-programme renvoie la nouvelle position. Le calcul de cette nouvelle position sera effectué par une fonction intermédiaire `nouvelle_position` à définir. Si le déplacement du joueur est impossible, le sous-programme `deplacement` renverra la position initiale.

```
function nouvelle_position (i j : int) (d : direction) : (int, int)
= (* à compléter *)
val deplacement (b : laby) (i j : int) (d : direction) : (int, int)
```

Partie II : programmation et preuve de correction

7. Programmer les trois sous-programmes précédents, ainsi que le programme testant l'égalité entre deux cases. Vous devez donc remplacer chaque spécification de la forme `val pgm …` par une définition de sous-programme de la forme `let pgm … = ….`
8. Démontrer que chacun des sous-programmes précédents vérifie sa spécification.

Partie III : initialisation

9. Spécifier le sous-programme `mk_laby` créant un labyrinthe valide quelconque.

```
val mk_laby (n : int) (m : int) : laby  (* à compléter *)
```

10. Spécifier le sous-programme `position_initiale` renvoyant une position initiale valide du joueur.

```
val position_initiale (b : laby) : (int, int) (* à compléter *)
```

11. Utiliser le module `LabyTest` pour tester au moyen d'assertions les deux sous-programmes précédents. Vous devrez compléter les tests déjà fournis.
12. Programmer le sous-programme `mk_laby` en remplissant de façon aléatoire les cases du labyrinthe créé, à l'aide des sous-programmes `random_bool` et `random_int`. Attention, un appel à `random_bool` s'écrit `random_bool ()`.
Avant d'écrire les invariants de boucle, il est fortement recommandé de dessiner sur une feuille le labyrinthe partiellement parcouru au bout d'un nombre quelconque d'itérations.

```
val random_bool () : bool
val random_int (n :int) : int
requires { 0 < n }
ensures { 0 <= result < n }
```

13. Démontrer que le sous-programme `mk_laby` vérifie sa spécification.
14. Programmer `position_initiale` en utilisant deux boucles `for` et une référence temporaire `trouve` initialisée à faux, indiquant si la position a été trouvée.

15. Démontrer que `position_initiale` vérifie sa spécification.