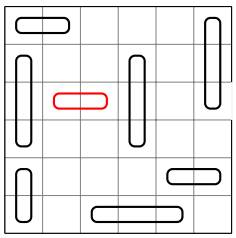
Le but du sujet est de modéliser le jeu rush hour (https ://fr.wikipedia.org/wiki/Rush_hour_(casse-tête)) qui est une sorte de jeu de taquin.

On dispose sur une grille de 6 cases par 6 cases (mais on peut imaginer une grille de taille différente), des voitures (de deux cases) et des camions (de trois cases) qui peuvent se déplacer en glissant uniquement dans un sens (vertical ou horizontal selon leur orientation initiale). Une des voitures est rouge et le but est de l'extraire par un coté ouvert de la grille. Si c'est possible on dit que la grille peut être résolue.



Premier exemple du jeu

On modélisera les véhicules par le type enregistrement

```
type vehicule =
 {taille  : int  ; orientation : char ; rouge : bool ; posFixe : int ; posMobile : int};;
```

où `taille` est un entier qui vaudra 2 pour une voiture et 3 pour un camion ; `orientation` est un caractère qui vaudra `'v'` si le véhicule est vertical et `'h'` s'il est horizontal ; `rouge` est un booléen selon que c'est le véhicule rouge ou non ; `posFixe` est un entier qui désigne la coordonnée du véhicule qui ne peut pas être modifié (l'abscisse si le véhicule est vertical et l'ordonné dans l'autre cas) et pour finir, `posMobile` est un entier qui désigne la coordonné du véhicule qui peut évoluer (on choisira la case la plus à gauche et la plus en bas du véhicule).
Pour la représentation graphique, la case $(0, 0)$ est en bas à gauche.
Dans la grille ci-dessus, le véhicule en bas à droite est `{taille = 3 ; orientation = 'h' ; rouge = false ; posFixe 0 ; posMobile = 2)}` alors que le véhicule rouge est `{taille = 2 ; orientation = 'h' ; rouge = true ; posFixe = 3 ; posMobile = 1}`
Une grille est un tableau de véhicules :

```
type grille = vehicule array;;
```

On considérera TOUJOURS que le véhicule rouge est dans la première case - d'indice 0 **- du tableau.**
L'exemple ci dessus est donc donné par

```
let v0 = {taille = 2 ; orientation = `h` ; rouge = true ; posFixe = 3; posMobile = 1};;
let v1 = {taille = 2 ; orientation = `h` ; rouge = false ; posFixe = 5; posMobile = 0};;
let v2 = {taille = 2 ; orientation = `h` ; rouge = false ; posFixe = 1; posMobile = 4};;
let v3 = {taille = 2 ; orientation = `v` ; rouge = false ; posFixe = 0; posMobile = 0};;
let v4 = {taille = 3 ; orientation = `h` ; rouge = false ; posFixe = 0; posMobile = 2};;
let v5 = {taille = 3 ; orientation = `v` ; rouge = false ; posFixe = 0; posMobile = 2};;
let v6 = {taille = 3 ; orientation = `v` ; rouge = false ; posFixe = 3; posMobile = 2};;
let v7 = {taille = 3 ; orientation = `v` ; rouge = false ; posFixe = 5; posMobile = 3};;

let g0 = [|v0;v1;v2;v3;v4;v5;v6 ; v7|];;
```

On va étudier les déplacements possibles à partir d'une grille donnée. On utilisera le type déplacement

```
type deplacement = {numVehicule : int ; augmente : int}
```

Le champ `numVehicule` désigne le numéro du véhicule dans la grille, le champ `augmente` est un entier valant ± 1 qui indique la modification du champ `posMobile` du véhicule. Par exemple, toujours pour la grille donnée en exemple ci-dessus, si on suppose que le véhicule numéroté 1 est celui en haut à gauche, le déplacement `nbVehicule = 1 ; augmente = 1` consiste à le déplacer vers la droite. De ce fait `posMobile` vaudra alors 1.
Pour finir on va gérer les modifications de la grille initiale dans un tableau `modif` tel que `modif.(i)` est un entier relatif qui désigne la modification de la position mobile du véhicule `i` par rapport à sa position initiale.
Si on déplace le véhicule `v1` de deux cases vers la droite et le véhicule `v5` de une case vers le haut, on aura `modif = [|0 ; 2 ; 0 ; 0 ; 0 ; 1 ; 0 ; 0 |]`.
Pour les tests, on pourra aussi utiliser la grille `g0bis` obtenue en remplaçant le véhicule `v2` par

```
let v2bis = {taille = 2 ; orientation = `v` ; rouge = false ; posFixe = 5; posMobile = 0};;
```

I - Fonctions utiles

Commençons par implémenter des fonctions qui nous servirons par la suite.

1) On se donne une grille `g0 = [|v0 ; ... ; vp|]` de départ avec $(p+1)$ véhicules. On veut alors associer un entier à chacune des grilles modifiées que l'on peut obtenir à partir de celle là. Comme la position mobile de chaque véhicule peut varier entre 0 et 4, on associe à chaque grille

$$\sum_{k=0}^{p} x_k 5^k = x_0 5^0 \times x_1 5^1 \times \cdots \times x_p 5^p$$

où x_i désigne la position mobile du véhicule `vi`.

   a) Écrire une fonction `grilleToNombre : grille -> int array -> int` telle que `grilleToNombre g0 modif` renvoie l'entier associée à la grille obtenue en effectuant les modifications données dans le tableau `modif` à la grille initiale `g0`.

   b) Écrire une fonction `nombreToGrille : grille -> int -> int array` telle que `nombreToGrille g0 n` renvoie le tableau des modifications à effectuer à partir de la grille `g0` pour obtenir une grille dont le nombre associé est `n`. On pourra penser à la décomposition de n en base 5.

2) Écrire une fonction `caseOccupee : grille -> int array -> int -> int*int -> bool` telle que `caseOccupee g0 modif i (x,y)` renvoie `true` si et seulement si le véhicule numéroté `i` dans la grille `g0` modifiée par le tableau `modif` occupe la case de coordonnée `(x,y)`.

3) Écrire une fonction `sortie : grille -> int array -> bool` telle que `sortie g0 modif` renvoie un booléen selon que la voiture rouge est ou non en position de sortir (c'est-à-dire qu'elle occupe la case $(5, 3)$) dans la grille déduite de `g0` par le tableau `modif`.

4) a) Écrire une fonction `deplacementValide : grille -> int array -> deplacement -> bool` telle que `deplacementValide g0 modif d` renvoie `true` si et seulement si, en partant de la position des véhicules stipulée dans la grille obtenue en modifiant la grille `g0` à l'aide du tableau `modif` on peut réaliser le déplacement `d`. On vérifiera que les véhicules ne s'interpénètrent pas et qu'un véhicule ne sort pas du cadre.

5) Écrire une fonction `deplacementsPossibles : grille -> int array -> deplacement list` telle que `deplacementsPossibles g0 modif` renvoie la liste des déplacements possibles à partir de la grille obtenue en appliquant les modifications `modif` à la grille `g0`.

6) Écrire une fonction `faitDeplacement : grille -> int array -> deplacement -> int array` qui renvoie le tableau des modifications obtenu après avoir fait le déplacement.

II - Parcours de graphe

On veut maintenant savoir si, partant d'une grille `g0`, il existe des déplacements qui permettent de faire sortir la voiture rouge. On va pour cela explorer tous les cas possibles jusqu'à trouver une éventuelle solution. On va réaliser un parcours du graphe dont les sommets sont les grilles que l'on peut obtenir et les arêtes sont données par les déplacements.
Notons que l'on n'utilisera pas directement les grilles mais les entiers associés à une grille comme fait à la question 1).

7) Écrire une fonction `admetSolutions : grille -> int` qui renvoie un entier valant -1 si la grille ne peut pas être résolue et le nombre associé à la grille où la voiture rouge est prête à sortir sinon. On pourra réaliser un parcours de graphe en largueur en implémentant les files à l'aide de référence de listes.

   On testera que la grille `g0` peut être résolue et que la grille `g0` ne peut pas l'être.

8) On cherche maintenant, à partir d'une grille `g0` qui peut être résolue, à obtenir la suite des déplacements qui permettent de faire sortir la voiture rouge. Pour cela on va définir un tableau d'entiers `precedent` avec 5^{p+1} cases. A la fin de l'execution de la fonction, on veut que si `nb` est un entier qui correspond à une grille atteinte lors de notre recherche, la case `precedent.(nb)` contienne le numéro de la grille **dont on est venu** quand on a atteint la grille de numéro `nb` **la première fois**.

   a) Modifier la fonction `admetSolutions` pour qu'elle renvoie le booléen comme ci-dessus ainsi que le tableau `precedent`.

   b) Écrire une fonction `solution : grille -> int list` qui renvoie une liste vide si la grille n'est pas résoluble et renvoie la liste des déplacements à faire dans le cas contraire.