Problème
Déplacement d'un cavalier sur un échiquier

Pour toutes les fonctions récursives, **on essaiera, dans la mesure du possible, de proposer une implémentation récursive terminale**.
Cependant, une version non terminale correcte sera toujours mieux notée qu'une version terminale incorrecte.
En cas de doute sur la version terminale, je vous invite à écrire une version non terminale puis, à proposer en dessous votre tentative de version terminale, pour sécuriser l'obtention de points.

Un échiquier est un plateau avec 8 lignes et 8 colonnes. Ces lignes et ces colonnes seront dans cet exercice numérotées de 0 à 7 à partir du coin supérieur gauche. Une position sur l'échiquier sera un couple (i, j) d'entiers naturels avec i le numéro de ligne et j le numéro de colonne.
Un cavalier placé sur l'échiquier se déplace en bougeant de deux cases dans une direction et de une case perpendiculairement. On représente ci-dessous à titre d'exemple les positions que peut atteindre un cavalier en un déplacement (marquées par un X) à partir d'une position initiale (marquée par un C) :

	0	1	2	3	4	5	6	7
0								
1			X		X			
2		X				X		
3				C				
4		X				X		
5			X		X			
6								
7								

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4	X		X					
5				X				
6		C						
7				X				

Une position (i, j) sera dite :

- **Valide** si elle appartient bien à l'échiquier.
- n-**Accessible** à partir de la position (k, l), $n \in \mathbb{N}$, si elle est valide et si le cavalier peut atteindre la position (i, j) à partir de la position (k, l) en **au minimum** n déplacements. Dans le cas $n = 1$ on dira que (i, j) est un **successeur** de (k, l).
  Ainsi la seule position 0-accessible à partir de (i, j) est (i, j), et les croix des tableaux précédents représentent les successeurs de la position C.
- **Accessible** à partir de la position (k, l) si elle est valide et si le cavalier peut l'atteindre à partir de la position (k, l) en un nombre fini de déplacements.

1. **Recherche des successeurs.**
   On considère ici et dans tout le reste de l'exercice que l'on a accès à une liste `dep` contenant les déplacements autorisés du cavalier :

```
# let dep=[(-2,-1);(-2,1);(-1,-2);(-1,2);(1,-2);(1,2);(2,-1);(2,1)];;
val dep : (int * int) list =
  [(-2, -1); (-2, 1); (-1, -2); (-1, 2); (1, -2); (1, 2); (2, -1); (2, 1)]
```

Où par exemple $(-1, 2)$ signifie que le cavalier passe (si c'est possible) d'une position (i, j) à une position $(i-1, j+2)$.

(a) Écrire une fonction `valide` de type `int * int -> bool` qui vérifie si une position est valide.

(b) Écrire une fonction `successeurs (i,j)` qui retourne la liste des successeurs de (i, j). Elle devra retourner la liste vide si (i, j) n'est pas valide.

(c) Écrire une fonction `flat` qui transforme en liste une liste de listes. Un résultat possible est par exemple :

```
# flat [ [] ; [(1,1)] ; [(1,1);(2,2)] ; [(1,1);(2,2);(3,3)] ];;
- : (int * int) list = [(1, 1); (1, 1); (2, 2); (1, 1); (2, 2); (3, 3)]
```

L'ordre des couples dans le résultat n'a aucune importance et ne doit pas nécessairement suivre celui de cet exemple.

(d) Écrire une fonction `liste_successeurs l` qui retourne la liste de tous les successeurs de toutes les positions d'une liste de positions `l`. Elle devra retourner la liste vide si `l` est vide et on ne cherchera pas à éliminer les doublons éventuels.

2. **Construction de la matrice d'accès.**
Dans toute cette partie (i, j) désignera une position valide quelconque.
Pour tout $n \in \mathbb{N}$, on note $M(n)$ la matrice de dimension 8x8 dont on numérote les lignes et les colonnes de 0 à 7 pour respecter la convention de l'échiquier et dont le coefficient en k^{eme} ligne et l^{eme} colonne vaut :

- p si la position (k, l) est p-accessible à partir de la position (i, j) avec $p \leq n$.
- -1 sinon.

On a par exemple à partir de la position $(0, 0)$ (en remplaçant les -1 par des $*$) :

$$M(0) = \begin{pmatrix} 0 & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{pmatrix}, \quad M(1) = \begin{pmatrix} 0 & * & * & * & * & * & * & * \\ * & * & 1 & * & * & * & * & * \\ * & 1 & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{pmatrix}$$

$$M(2) = \begin{pmatrix} 0 & * & 2 & * & 2 & * & * & * \\ * & * & 1 & 2 & * & * & * & * \\ 2 & 1 & * & * & 2 & * & * & * \\ * & 2 & * & 2 & * & * & * & * \\ 2 & * & 2 & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \\ * & * & * & * & * & * & * & * \end{pmatrix}$$

Enfin on note M la matrice de dimension 8x8 dont le coefficient en k^{eme} ligne et l^{eme} colonne (numérotées de 0 à 7) vaut p si la position (k, l) est p-accessible à partir de la position (i, j), -1 sinon.

Voici quelques commandes permettant de créer/modifier des matrices (qui sont en fait des tableaux de tableaux) en OCaml :

- On crée une matrice de dimension `nxp` dont les coefficients sont tous égaux à *expr* avec l'instruction `Array.make_matrix n p` *expr* :

```
# let m=Array.make_matrix 2 4 (-1);;
val m : int array array = [|[|-1; -1; -1; -1|]; [|-1; -1; -1; -1|]|]
```

- On accède à l'élément de la matrice `m` situé en ligne `n` et en colonne `p` avec `m.(n).(p)`.
- On modifie l'élément de la matrice `m` situé en ligne `n` et en colonne `p` avec `m.(n).(p)<-`*expr*.

```
# m.(1).(3)<-8;;
- : unit = ()
# m;;
- : int array array = [|[|-1; -1; -1; -1|]; [|-1; -1; -1; 8|]|]
```

(a) Prouver qu'il existe un entier $N \in \mathbb{N}$ tel que pour toute position valide (i, j) et pour tout $n \in \mathbb{N}$, $n \geq N \Rightarrow M(n) = M$.

(b) Écrire une fonction `transition l m p` qui, lorsque `l` est la liste des successeurs des positions $(p-1)$-accessibles à partir d'une position initiale et lorsque `m` est la matrice $M(p-1)$, retourne la liste des positions p-accessibles et modifie `m` afin qu'elle soit égale à $M(p)$.

(c) Écrire une fonction `cavalier (i,j)` qui retourne la matrice M obtenue à partir de la position initiale (i, j). On supposera que (i, j) est une position valide.

(d) Justifier que `cavalier` se termine toujours en temps fini.

3. **Recherche de positions inaccessibles.**
On cherche à savoir si toute position est accessible à partir de toute position initiale. Plutôt que chercher à le prouver mathématiquement (pas très difficile mais pénible à cause d'un certain nombre de cas particuliers), on va le vérifier à l'aide d'un programme.
Pour cette question et uniquement pour cette question, on pourra utiliser des instructions `for` ou `while`.

Écrire une fonction `test` de type `unit -> bool` qui retourne `true` s'il existe une position inaccessible à partir d'au moins une position initiale, `false` sinon. *Après avoir proposé une approche exhaustive, on pourra proposer, en justifiant par des arguments mathématiques, une approche permettant de limiter le nombre de tests effectués.*

Corrigé de l'exercice 3.

[Retour à l'énoncé]

1. Recherche des successeurs.

a. Une position (i, j) est valide si et seulement si $-1 < i < 8$ et $-1 < j < 8$, on a donc immédiatement :

```
let valide (i,j) = -1<i && i<8 && -1<j && j<8;;
```

b. On commence par écrire une fonction `add (i,j) (a,b)` qui retourne la position obtenue à partir de la position `(i,j)` avec le déplacement `(a,b)` :

```
let add (i,j) (a,b) = (i+a,j+b);;
```

On écrit alors une fonction auxiliaire récursive `successeurs_aux (i,j) d r` où `(i,j)` est une position valide, `d` une liste de déplacements et `r` une liste accumulateur qui contiendra à la fin de l'exécution la liste des successeurs de `(i,j)` :

- Si `d` est vide, il n'y a plus de déplacements à tester et on retourne `r`.
- Si `d=h::t` alors on regarde la position `suiv` obtenue à partir de `(i,j)` avec le déplacement `h` : `suiv=add (i,j) h`. Si cette position est valide, on l'ajoute à la liste des successeurs et on poursuit l'algorithme sur `t` : `successeurs_aux (i,j) t (suiv::r)`. Si cette position n'est pas valide, on ne l'ajoute pas à la liste des successeurs et on poursuit l'algorithme sur `t` : `successeurs_aux (i,j) t r`.

```
let rec successeurs_aux (i,j) d r = match d with
    |[] -> r
    |h::t -> let suiv=add (i,j) h in
               if valide suiv then successeurs_aux (i,j) t (suiv::r)
               else successeurs_aux (i,j) t r;;
```

Pour programmer `successeurs (i,j)`, il suffit d'appeler `successeurs_aux (i,j) dep []` si `(i,j)` est valide ou de renvoyer la liste vide si `(i,j)` n'est pas valide :

```
let successeurs (i,j) =
    if valide (i,j) then  successeurs_aux (i,j) dep []
    else [];;
```

c. Une première version, non récursive terminale de `flat`donne

```
let rec flat l = match l  with
    |[] -> []
    |[]::l' -> flat l'
    |(t::q)::l' -> t::(flat (q::l'));;
```

Une version terminale :

```
let flat ll =
  let rec aux ll acc =
    match ll with
    | [] -> acc
    | l::qll -> match l with
                 | [] -> aux qll acc
                 | h::q -> aux (q::qll) (h::acc)
   in
   aux ll [] (* pas le même ordre dans la liste retour que l'énoncé...
                mais pas  du tout indispensable *)
;;
```

d. On commence par écrire à l'aide de la fonction `successeurs` une fonction récursive `liste_successeurs_aux` qui retourne la liste des listes des successeurs :

```
let rec liste_successeurs_aux l res=match l with
    |[] -> res
    |h::t -> (successeurs h)::(liste_successeurs_aux t res);;
```

Il ne reste alors plus qu'à "l'aplatir" avec la fonction `flat` :

```
let liste_successeurs l = flat (liste_successeurs_aux l []);;
```

Une version terminale avec encapsulation :

```
let liste_successeurs l =
  let rec aux l acc =
    match l with
    | [] -> acc
    | (i,j)::q -> aux q ((successeurs i j):: acc)
  in
  flat (aux l [])
;;
```

2. Construction de la matrice d'accès.

a. Il y a $8 \times 8 = 64$ cases distinctes sur l'échiquier. Montrons alors que si une case est accessible à partir de (i,j), elle l'est au plus en 63 déplacements.

Soit (i,j) une position valide et soit (k,l) une position accessible à partir de (i,j). Un chemin de (i,j) à (k,l) sera une suite de positions valides $(p_0, p_1, \cdots, p_N)$ vérifiant :

- $p_0 = (i,j)$ et $p_N = (k,l)$.
- Pour tout $i \in [\![0, N-1]\!]$, p_{i+1} est un successeur de p_i.
- Pour tout $i \in [\![1, N-1]\!]$, p_{i+1}, $p_i \neq p_0$ et $p_i \neq p_N$. Cette dernière condition n'est pas nécessaire mais permettra de simplifier la rédaction par la suite.

Si $c = (p_0, p_1, \cdots, p_N)$ est un chemin de (i,j) à (k,l), la longueur de c (notée $|c|$) sera N.

(k,l) étant accessible à partir de (i,j), l'ensemble des chemins de (i,j) à (k,l) est non-vide et soit alors $c = (p_0, p_1, \cdots, p_N)$ un chemin de longueur minimale N. Montrons alors par l'absurde que $N \leq 63$.
Si $N > 63$ alors $N - 1 > 62$ et on déduit alors de la troisième condition qu'il existe deux entiers i et j vérifiant $1 \leq i < j \leq N-1$ et tels que $p_i = p_j$. Alors $(p_0, \cdots p_i, p_{j+1}, \cdots, p_N)$ est un chemin de (i,j) à (k,l) de longueur strictement inférieur à N ce qui contredit l'hypothèse de minimalité de c.
Pour toute position valide (i,j), les positions accessibles à partir de (i,j) le sont au plus en 63 déplacements et donc pour tout $n \in \mathbb{N}$, $n \geq 63 \Rightarrow M(n) = M$.

b. Les positions p-accessibles à partir d'une position initiale sont à chercher parmi les successeurs des positions $(p-1)$-accessibles à partir de la position initiale. Il faut simplement vérifier qu'ils ne sont pas accessibles en moins de p mouvements, leur coefficient dans la matrice $M(p-1)$ doit donc être -1.

Comme précédemment, on commence par coder une fonction auxiliaire récursive `transition_aux l m p res`, la liste `res` servant d'accumulateur pour stocker le résultat.

- Si la liste `l` est vide, il n'y a plus de positions à tester et on retourne `res`.
- Si `l=(a,b)::q`, on regarde si la position `(a,b)` n'a pas déjà été atteinte précédemment, c'est-à-dire si `m.(a).(b)= -1`.
  Si ce n'est pas le cas, `(a,b)` n'est pas p-accessible et on ne la conserve pas et on appelle alors `transition_aux` sur `q` : `transition_aux q m p res`.
  Si c'est le cas, `(a,b)` est p-accessible. On conserve cette information en modifiant la matrice `m` en conséquence : `m.(a).(b) <- i` puis on ajoute `(a,b)` à `res` et on appelle alors `transition_aux` sur `q` : `transition_aux q m p ((a,b)::res)`.

```
let rec transition_aux l m p res = match l with
    |[] -> res
    |(a,b)::q ->
        if (m.(a).(b)= -1) then
            begin
                m.(a).(b) <- p;
                transition_aux q m p ((a,b)::res);
```

```
            end
        else transition_aux q m p res;;
```

Pour programmer `transition l m p`, il suffit d'appeler `transition_aux l m p []` :

```
let transition l m p = transition_aux l m p [];;
```

c. On va se servir de la fonction `transition` pour construire pas à pas la matrice M. Pour cela on commence par écrire une fonction récursive auxiliaire `cavalier_aux l m p` qui, lorsque `l` est la liste des successeurs des positions $p-1$-accessibles et lorsque `m` est la matrice $M(p-1)$, retourne la matrice M.

- Si `l` est vide alors il n'y a pas de position p-accessible et alors $M(p-1) = M$, il suffit de renvoyer `m`.
- Si `l` n'est pas vide alors il faut appeler `cavalier_aux` avec : la liste des successeurs des positions p-accessibles, la matrice $M(p)$ et l'entier $p+1$.
  Pour réaliser cela on utilise la fonction `transition` et on stocke son résultat dans une liste `ls = liste_successeurs(transition l m p)`. Alors `ls` est bien la liste des successeurs des positions p-accessibles et `m` a été modifiée et vaut désormais $M(p)$.

```
let rec cavalier_aux l m p = match l with
    |[] -> m
    |_  -> let ls = liste_successeurs(transition l m p) in cavalier_aux ls m (p+
```

Pour coder `cavalier`, il suffit (si `(i,j)` est valide) d'appeler `cavalier_aux [(i,j)] m 0` avec `m` une matrice 8×8 dont tous les coefficients valent 1 :

```
let cavalier (i,j)=
    let m=Array.make_matrix 8 8 (-1) in
    if valide (i,j) then
     cavalier_aux [(i,j)] m 0
    else
     failwith "Position initiale non compatible";;
```

d. On a vu à la question 2.a que quelle que soit la position initiale (i, j), les positions accessibles le sont en au plus 63 déplacements. Il y a donc au maximum 63 appels dans la fonction `cavalier`, elle termine donc bien en temps fini.

3. Recherche de positions inaccessibles.

Il y avait une erreur dans l'énoncé, la fonction `test` doit bien entendu retourner un booléen.
Une première idée pourrait être de calculer pour toute position valide (i, j) la matrice M et vérifier qu'elle ne contient pas de -1. Cette méthode contiendrait alors beaucoup de tests inutiles. En effet la propriété d'accessibilité est symétrique et transitive ; ainsi s'il existe une position à partir de laquelle toute position est accessible, alors toute position est accessible à partir de n'importe quelle autre position.
Il suffit donc de parcourir une matrice M et de vérifier si elle contient des -1 ou non. On choisit une position initiale "centrale" ce qui devrait intuitivement réduire le temps de calcul de M :

```
let test () =
  let ret = ref true in
  let m = cavalier (3,3) in
  let i = ref 0 in
  while (!i < 8 && !ret ) do
    let j = ref 0 in
    while (!j < 8 && !ret ) do
      if (m.(!i).(!j) = -1) then
        ret := false
      else
        j := !j +1;
    done;
    i := !i + 1;
  done;
  !ret
;;
(* appel de la fonction, sans arguments *)
test ();;
```