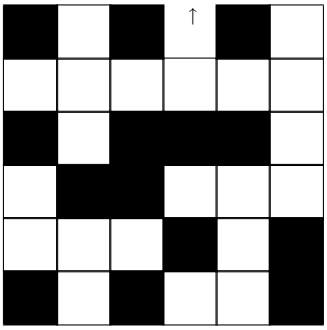
Feuille d'exercices n°14 de l'option d'informatique.

Thème : parcours dans un labyrinthe.

I. Introduction.

On considère un labyrinthe matérialisé par un damier carré de n^2 cases blanches ou noires. Une case blanche particulière, située au bord, symbolise la sortie.



Un pion est placé au départ sur une case blanche choisie par l'utilisateur.

À chaque coup, le pion peut se déplacer sur l'une des quatre cases voisines (à l'Est ou au Nord ou à l'Ouest ou au Sud), à condition que cette case soit permise, ce qui veut dire :

- qu'elle est blanche,
- qu'elle n'est pas située hors du damier,
- qu'elle n'a pas déjà été "visitée".

Le but du programme est de déterminer s'il existe au moins un chemin conduisant du point de départ choisi vers la sortie, et accessoirement de visualiser un tel chemin.

Le damier (qu'on pourra facilement créer ou modifier avec un éditeur de texte) se trouve stocké dans un fichier-texte nommé *labyr.dta* ayant n lignes, chaque ligne contenant exactement n caractères, chacun d'eux étant soit un 'M' pour une case noire, soit un 'O' pour une case blanche autre que la sortie, soit un (espace) pour la case blanche correspondant à la sortie.

```
M O M   M O
O O O O O O
M O M M M O
O M M O O O
O O O M O M
M O M O O M
```

La procédure *lit_labyrinthe* (qui sera fournie, mais qu'importe de comprendre) se charge de lire le fichier et de la transformer en une matrice carrée d'ordre $n+2$, le damier initial étant agrandi en un damier dont toutes les cases du bord sont considérées comme n'étant pas blanches (elles contiennent le caractère '1' par ex.). Cette astuce facilitera la programmation, le pion ne pouvant pas de ce fait se déplacer dans une case située en dehors du damier, une telle case n'étant pas blanche.

```
1 1 1 1 1 1 1 1
1 M O M   M O 1
1 O O O O O O 1
1 M O M M M O 1
1 O M M O O O 1
1 O O O M O M 1
1 M O M O O M 1
1 1 1 1 1 1 1 1
```

Remarques :

- les éléments du damier "intérieur", qui sont des caractères sont repérés par leur position (i, j) avec $1 \leq i \leq n$ et $1 \leq j \leq n$, le coin supérieur gauche ayant l'indice $(1, 1)$ comme en calcul matriciel.
- une matrice étant un objet Caml **mutable**, il sera possible de modifier certaines valeurs, notamment pour signaler qu'un case a déjà été visitée.

II. Algorithme itératif utilisant une pile.

Cet algorithme sera décrit ici de façon très informelle : le travail restant à faire consiste à le traduire en un programme en Caml.

On utilisera ici une pile représentée par une liste Caml dont les éléments sont des couples (i, j) d'entiers. Ces couples représentent, à un moment donné, l'ensemble des positions sur le damier qui ont été rencontrées comme étant à envisager éventuellement et qui sont en attente de l'être. En effet, lorsque le pion arrive à un carrefour où il a le choix par exemple entre deux directions possibles, il va en essayer une sur les deux et empiler l'autre de façon à la conserver en mémoire et pouvoir l'essayer plus tard à son tour si la première n'aboutit pas. On suppose qu'on part de la position initiale (i_0, j_0) choisie. Cette position est immédiatement placée sur la pile.

Tant que la sortie n'a pas été trouvée et que la pile n'est pas vide :

- on dépile le couple (i, j) situé au sommet de la pile (ie en tête de liste).
- si cette position est celle de la sortie, c'est gagné!
- sinon :
  - ∗ on empile les positions voisines situées à l'Est, au Nord, à l'Ouest et au Sud lorsqu'elles sont «blanches» (ie contiennent 'O' ou ' ').
  - ∗ on «noircit» avec un 'V' la case (i, j) que l'on vient de dépiler pour indiquer qu'elle a été visitée, afin de ne plus la prendre en compte ultérieurement.

Fin du tant que.

Si a un moment donné, la pile est vide, c'est que tous les cas possibles ont été envisagés sans succès.

Il est fortement conseillé, pour être certain d'avoir bien compris, d'appliquer "à la main" l'algorithme sur le damier de la page 1 en partant de la position (5, 5) et on observant le déroulement des 6 premières itérations et les contenus successifs de la pile.

Remarque importante : lorsque le pion aura pu trouver la sortie, toutes les cases visitées seront remplies avec le caractère 'V' : il n'est pas possible de distinguer celles qui ont été examinées inutilement. Autrement dit, on est certain qu'il existe au moins un chemin joignant (i_0, j_0) à la sortie, mais on ne peut pas savoir lequel si l'on n'introduit pas des structures de données supplémentaires servant à mémoriser les divers essais effectués et "à remonter" à partir de la sortie une fois celle-ci trouvée.

Facultatif : Imaginer une méthode permettant de reconstituer le chemin, celui-ci étant alors matérialisé par des cases contenant le symbole '+' à la place des 'V'.

III. Algorithme récursif.

Principe : Un chemin joignant la position (i, j) à la sortie est soit un chemin de longueur nulle lorsque (i, j) est la position de la sortie, soit la "réunion" de (i, j) et d'un chemin joignant une des cases voisines permises de (i, j) à la sortie.
On positionne un booléen appelé ok à *false*.
Lorsque le pion se trouve en position (i, j) :

- si c'est la position de sortie, on met ok à true.
- Sinon, on commence par placer le caractère 'V' en position (i, j) pour empêcher le pion d'y revenir. Puis on examine récursivement les cases voisines qui sont permises, par ex. dans l'ordre E, N, O, S. Lorsqu'une de ces cases est le point de départ d'un chemin aboutissant à la sortie, il est alors inutile d'examiner les suivantes éventuelles. De plus, on peut remplacer le caractère 'V' se trouvant en par le caractère '+' pour exprimer (i, j) se trouve sur le chemin final trouvé.

IV. Programmation

Il s'agit de compléter chacun des deux programmes suivants dans lesquels seules les procédures correspondant à la recherche d'un chemin vers la sortie à partir d'une position (blanche) (i_0, j_0) initiale ont été laissées en blanc.

Pour chacun des deux programmes, on utilise les mêmes fonctions pour créer la matrice, l'afficher ou la modifier :

```
let n=10;;      (*  pour un damier 10x10 *)
let lit_labyrinthe nom_fichier =
   let lab = make_matrix (n+2) (n+2) 'l'
       and fichier = open_in nom_fichier and ligne = ref("") in
       for i=1 to n do
           ligne := input_line fichier;
           for j=1 to n do lab.(i).(j) <- !ligne.[j-1] done;
       done;
   lab;;

let imprime lab =
    for i=1 to n do
        for j=1 to n do print_char(lab.(i).(j)) done;
        print_newline();
    done;;

let modifie lab i j c  = lab.(i).(j) <- c;;
```

1) Programme itératif.

```
let pile = ref [];;

let empile p x =       (* À compléter de type 'a list ref -> 'a ->unit = <fun> *)

let depile p =         (* À compléter de type 'a list ref -> 'a = <fun>)


let directions p lab i j =
   (* empilage des positions suivantes permises *)
                              (* À compléter *)

let sortie_trouvée lab i0 j0 =            (* renvoie true ou false *)
                                              (* A compléter *)
```

2) Programme récursif.

```
let sortie_trouvée lab i0 j0 =   (*  renvoie true ou faise *)
                                 (* utilise une fonction locale récursive *)
                                 (* À compléter *)
```

Ainsi, les deux fonctions ont le même nom dans chacun des deux programmes (indépendants). On peut donc utiliser la même fonction que voici pour les appeler

```
let essai lab i0 j0 =
       if lab.(i0).(j0) = '1' or lab.(i0).(j0) = 'M'
         then print_string("Point de départ non permis")
         else
             begin
                if sortie_trouvée lab i0 j0
                      then print_string("J'ai trouve la sortie")
                      else print_string("Je suis enfermé !");
                print_newline();
                imprime lab;
             end;;
```

On pourra ensuite tester chacun de ces deux programmes en tapant par exemple chaque fois :

```
let lab = lit_labyrinthe "labyr.dta";;
essai lab 9 8;;
```

On doit en effet régénérer la matrice initiale puisque les programmes la modifient.