Exercice 1 : Robot

On souhaite simuler le déplacement d'un robot sur un plateau rectangulaire contenant éventuellement des points à collecter. Le robot peut se déplacer, changer de direction et ramasser automatiquement les points présents sur la case où il se trouve.

Le plateau. Le plateau est une grille rectangulaire composée de cellules. Chaque cellule peut :
- être vide ;
- contenir un certain nombre de points.

```
type cellule =
  | Rien                    (* Une cellule est vide ...   *)
  | Point of int            (* ... ou contient des points *)

type plateau = cellule array array
```

Le plateau est représenté par un tableau bidimensionnel.

```
type plateau = cellule array array
```

Le robot. Un robot est caractérisé par :
- sa position (i, j) sur le plateau (i : numéro de ligne, j : numéro de colonne)
- son orientation (direction vers laquelle il regarde).

Les directions possibles sont : Nord (N) Sud (S) Est (E) Ouest (O).

```
type direction =
  | N | S | E | O

type robot =
  {
    mutable i : int             ; (* La ligne du robot *)
    mutable j : int             ; (* La colonne du robot *)
    mutable attitude : direction ; (* Où regarde le robot ? *)
  }
```

Les ordres. Le robot peut recevoir une liste d'ordres, les ordres étant de deux types :
- Avance : le robot avance d'une case dans la direction où il regarde ;
- Tourne toi dans la direction d : le robot change son orientation pour regarder dans la direction d.

Collecte de points Après chaque ordre exécuté :
- le robot ramasse automatiquement les points présents sur sa case ;
- ces points sont ajoutés à son score total ;
- la cellule devient alors vide.

Si le robot sort du plateau, une erreur doit être signalée.

Q. 1 Écrire une fonction `tourne : robot -> direction -> unit` prenant en paramètre un robot et le modifiant pour qu'il regarde dans la direction qui lui est passée en paramètre.

Q. 2 Écrire une fonction `delta : direction -> int * int` qui retourne : $(-1, 0)$ dans le cas `N`, $(1, 0)$ dans le cas `S`, $(0, 1)$ dans le cas `E`, $(0, -1)$ dans le cas `O`.

Q. 3 En déduire une fonction `avance : robot -> unit` prenant en paramètre un robot et le modifiant pour qu'il avance d'une case dans la direction dans laquelle il regarde.

Q. 4 Écrire une fonction `execute_ordre : robot -> ordre -> unit` prenant en paramètre un robot et un ordre et faisant exécuter l'ordre par le robot.

Q. 5 Écrire une fonction `mange_point : robot -> plateau -> int` prenant en paramètre un robot et lui faisant vider les points de la case sur laquelle il se trouve. On renverra le nombre de points ainsi collectés.

Q. 6 Écrire une fonction `execute_robot : robot -> plateau -> ordre list -> int` prenant en paramètres un robot, un plateau et un liste d'ordres pour le robot et retournant la somme des points collectés par le robot lorsqu'il effectue la liste d'ordres. On devra modifier par effets de bord le robot ainsi que le plateau.

Solution

```
let tourne (r: robot) (d: direction) : unit =
  r.attitude <- d

let delta (d: direction): int * int =
  match d with
  | N -> (-1,  0)
  | S -> ( 1,  0)
  | E -> ( 0,  1)
  | O -> ( 0 ,-1)

let avance (r: robot): unit =
  let di, dj = delta r.attitude in
  r.i <- r.i + di ;
  r.j <- r.j + dj

let execute_ordre (r: robot) (o: ordre): unit =
  match o with
  | Avance   -> avance r
  | Tourne d -> tourne r d

let mange_point (r: robot) (p: plateau): int =
  let n = Array.length p     in
  let m = Array.length p.(0) in
  if r.i < 0 || r.j < 0 || r.i >= n || r.j >= m then failwith "hors du plateau"
  else
    match p.(r.i).(r.j) with
    | Point(pp) -> begin
        p.(r.i).(r.j) <- Rien ;
        pp
      end
    | Rien -> 0

let execute_robot (r: robot) (p: plateau) (os: ordre list): int =
  (* Exécute la liste d'ordre os, points_accumules est la somme des points
     totalisés jusqu'à présent. *)
  let rec aux (points_accumules: int) (os: ordre list): int =
    match os with
    | []       -> points_accumules
    | o :: os' ->
       execute_ordre r o ;
       let pp = mange_point r p in
```

```
        aux (pp + points_accumules) os'
    in aux 0 os
```

Exercice 2 : Suffixe

On dit d'une liste `l` qu'elle est un suffixe d'une liste `l'` dès lors qu'il existe une liste `r` telle que `l'` est `r @ l`.

Q. 1 Définir une fonction `suffixe_au_plus_n : 'a list -> int -> 'a list * int` prenant en paramètres une liste `l` et un entier `n` et retournant le plus long suffixe de `l`, de longueur au plus `n` ainsi que la taille de ce suffixe. Par exemple : `suffixe_au_plus_n [0; 1; 2; 3; 4; 5] 3 = ([3; 4; 5], 3)`, mais `(suffixe_au_plus_n [0; 1; 2; 3; 4; 5] 10) = ([0; 1; 2; 3; 4; 5], 6)`.

Q. 2 Définir une fonction OCaml `suffixe : 'a list -> 'a list -> bool` prenant en paramètres deux listes `l` et `l'` et testant si la liste `l` est un suffixe de la liste `l'`.

Solution

```
(* Calcule le plus long suffixe de l de longueur au plus n et retourne aussi sa longueur.
↪  *)
let rec suffixe_au_plus_n (l: 'a list) (n: int): 'a list * int =
  match l with
  | []     -> ([], 0)
  | x :: r ->
```

Exercice 3 : Préfixe

On dit d'une liste `l` qu'elle est un préfixe d'une liste `l'` dès lors qu'il existe une liste `r` telle que `l'` est `l @ r`.

Q. 1 Définir une fonction OCaml `prefixe : 'a list -> 'a list -> bool` prenant en paramètres deux listes `l` et `l'` et testant si la liste `l` est un préfixe de la liste `l'`.

Solution

```
let rec prefixe (l: 'a list) (l': 'a list): bool =
  match l, l' with
  | [], _           -> true
  | x :: r, y :: r' -> (x = y) && prefixe r r'
  | _               -> false
```

Exercice 4 : Chunk

Q. 1 Définir une fonction OCaml `chunk : 'a list -> int -> 'a list list` prenant en paramètre une liste `l` et un entier `n` et retournant la liste des listes obtenues en découpant la liste initiale tous les `n` éléments. Par exemple `chunk [0; 1; 2; 3; 4; 5; 6; 7; 8; 9; 10; 11] 5 = [[0; 1; 2; 3; 4]; [5; 6; 7; 8; 9]; [10; 11]]`.

Solution

```
(* Retourne un couple constitué des n premiers éléments de l et du reste
   de la liste l. *)
let rec n_premiers_et_reste (l: 'a list) (n: int): 'a list * 'a list =
    match n, l with
    | 0, _ | _, [] -> [], l
    | _, x :: l'   ->
      let c, l'' = n_premiers_et_reste l' (n - 1) in
      (x :: c), l''

let rec chunk (l: 'a list) (n: int): 'a list list =
  match l with
  | [] -> []
  | _  ->
    let c, l' = n_premiers_et_reste l n in
    c :: (chunk l' n)
```

Exercice 5 : Répétition

Q. 1 Définir une fonction `repete : int list -> int array` prenant en paramètre une liste `l` d'entiers *naturels* et retournant le tableau dont les éléments sont les éléments de la liste, dans le même ordre, mais répétés autant de fois que leur valeur. Par exemple `(repete [0; 1; 2; 3; 3]) = [|1; 2; 2; 3; 3; 3; 3; 3; 3|]`

Exercice 6 : Test de sous-tableaux

On dit d'un tableau t de longueur n que c'est un *facteur* d'un tableau t' de longueur m dès lors qu'il existe $i \in [\![0, m-n]\!]$ tel que $\forall j \in [\![0, n-1]\!]$, $t'[i+j] = t[j]$.

Q. 1 Écrire une fonction `sous_tableau : 'a array -> 'a array -> int option` prenant en paramètres un tableau t et un tableau t' et testant si t est un facteur de t'. Si ce n'est pas le cas, la fonction retournera `None`, dans le cas contraire elle devra retourner `Some(i)` où `i` est un indice tel que $\forall j \in [\![0, n-1]\!]$, $t'[i+j] = t[j]$. Par exemple : `(sous_tableau [|1; 2; 3|] [|1; 2; 1; 2; 3|]) = Some(2)`, mais `(sous_tableau [|1; 2; 3|] [|1; 2; 1; 2|]) = None`.