Exercice 1 : Miroir d'un tableau

Q. 1 Écrire une fonction `miroir_a_to_l : 'a array -> 'a list` prenant en paramètre un tableau et retournant le miroir du tableau, sous forme de liste. Par exemple `miroir [|0; 1; 2; 3; 4|] = [4; 3; 2; 1; 0]`.

Solution

```
let miroir_a_to_l (t: 'a array): 'a list =
  let n = Array.length t in
  let res = ref [] in
  for i = 0 to (n-1) do
    res := t.(i) :: !res
  done; !res
```

Q. 2 Écrire une fonction `miroir_l_to_a : 'a list -> 'a array` prenant en paramètre une liste et retournant le miroir de cette liste, sous forme de tableau. Par exemple `miroir [0; 1; 2; 3; 4] = [|4; 3; 2; 1; 0|]`.

Solution

```
(* Calcule la longueur d'une liste *)
let rec len (l: 'a list): int =
  match l with
  | []     -> 0
  | _ :: r -> 1 + (len r)

let miroir_l_to_a (l: 'a list): 'a array =
  match l with
  | []     -> [||]
  | x :: _ ->
     let n = len l in
     let rep = Array.make n x in
     (* Itère sur la liste l et écrit son contenu dans rep, dans l'ordre
        inverse. *)
     let rec aux (l: 'a list) (i: int): unit =
       match l with
       | []     -> ()
       | x :: r -> rep.(i) <- x; aux r (i - 1)
     in
     aux l (n - 1)
```

Exercice 2 : Transposition

On considère des tableaux de taille n de listes d'entiers de $[\![0, n-1]\!]$. On appelle *transposé* d'un tel tableau t, un tableau t' de taille n, contenant dans sa case $i \in [\![0, n-1]\!]$ une liste telle que $\forall j \in [\![0, n-1]\!], j \in t[i] \Leftrightarrow i \in t'[j]$.

Q. 1 Définir une fonction `transpose : int list array -> int list array` calculant la transposée du tableau qui lui est passé en argument.

Solution

```
let transpose (t: int list array): int list array =
  let n = Array.length t in
  let res = Array.make n [] in
  Array.iteri (fun i l ->
      List.iter (fun j ->
          res.(j) <- i :: res.(j)
        ) l
    ) t ;
  res
```

Exercice 3 : Liste d'éléments à tableau indicateur

Soit une constante entière $M \in \mathbb{N}$ fixée ($M = 5$ dans les exemples). On considère deux représentations en OCaml des sous-ensembles de $[\![0, M-1]\!]$ (par exemple l'ensemble $\{0, 2, 3\}$).

a) La première au moyen d'une liste finie des éléments de l'ensemble (`[0; 2; 3]` ou `[3; 2; 0]` pour l'exemple ci-dessus).

b) La seconde au moyen d'un tableau de taille M de booléen dont la case $i \in [\![0, M-1]\!]$ indique si oui ou non l'élément est dans l'ensemble (`[|true; false; true; true; false|]` pour l'exemple ci-dessus).

Q. 1 Donner des fonctions de conversion `a_to_b : int -> int list -> bool array` et `b_to_a : bool array -> int list` d'un type vers l'autre. La fonction `a_to_b` prend en paramètres non seulement la liste en question mais aussi l'entier M de l'énoncé.

Solution

Version avec itérateur.

```
let a_to_b (m: int) (s: int list): bool array =
  let res = Array.make m false in
  List.iter (fun i -> res.(i) <- true) s ;
  res
```

Version sans itérateur.

```
let a_to_b (m: int) (s: int list): bool array =
  let res = Array.make m false in
  (* Itère sur la liste s et met à true, dans res, les indices se trouvant dans s. *)
  let rec aux (s: int list): unit =
    match s with
    | []      -> ()
    | i :: s' -> res.(i) <- true; aux s' in
  aux s; res
```

```
let b_to_a (a: bool array): int list =
  let n = Array.length a in
  (* Calcule une représentation de type a. pour le sous-tableau
     indicateur a[i:len(a)]. *)
  let rec aux (i: int): int list =
    if i = n then []
    else (if a.(i) then [i] else []) @ aux (i + 1)
  in aux 0
```

Exercice 4 : Polynômes creux

On considère dans cet exercice des polynômes de $\mathbb{Z}[X]$. On représente de tels polynômes en OCaml au moyen d'une liste de couples d'entiers : la liste `[(`a_0`, `d_0`); (`a_1`, `d_1`); (`a_2`, `d_2`); …; (`a_p`, `d_p`)]` avec pour tout $i \in [\![0, p]\!]$, $a_i \in \mathbb{Z}^\star$ et $d_i \in \mathbb{N}^\star$. Une telle liste représente alors le polynôme :

$$X^{d_0}(a_0 + X^{d_1}(a_1 + X^{d_2}(a_2 + \dots X^{d_{p-1}}(a_{p-1} + a_p X^{d_p}))))$$

On définit donc le type suivant.

```
type poly_1 = (int * int) list
```

On considère par ailleurs la représentation "classique" d'un polynôme $\sum_{i=0}^{n} b_i X^i$ au moyen d'un tableau de ses $n+1$ coefficients : `[|`b_0`; `b_1`; …; `b_n`|]`.
On définit donc le type suivant.

```
type poly_2 = int array
```

Q. 1 Donner des fonctions de conversion permettant la traduction d'une représentation vers l'autre. Discuter de la pertinence de l'une vis-à-vis de l'autre.

Solution

```
type poly_1 = (int * int) list
type poly_2 = int array

(** Calcule le degré du polynôme [p]. Version [List.iter]. *)
let calcul_degre (p: poly_1): int =
  let rep = ref 0 in
  List.iter (fun (_, d) -> rep := d + !rep) p;
  !rep
(** Calcule le degré du polynôme [p]. Version [List.fold_left]. *)
let calcul_degre (p: poly_1): int =
  List.fold_left (fun rep (_, d) -> d + rep) 0 p

(** Conversion d'un [poly_2] vers un [poly_1] *)
let poly_2_depuis_poly_1 (p: poly_1): poly_2 =
  let deg_p   = calcul_degre p in
  let rep     = Array.make (deg_p + 1) 0 in
  let deg_cur = ref 0 in
  List.iter (fun (a, d) ->
      deg_cur := !deg_cur + d;
      rep.(!deg_cur) <- a
    ) p;
  rep

(** Conversion d'un [poly_1] vers un [poly_2]. *)
let poly_1_depuis_poly_2 (p: poly_2): poly_1 =
  let n = Array.length p in
  (** Calcule une représentation sous forme de [poly_1] du polynôme
      représenté par les cases d'indices >= i du tableau p
      multiplié par (X^deg_courant) *)
  let rec reste (deg_courant: int) (i: int) =
    if i >= n then []
    else if p.(i) <> 0 then (p.(i), deg_courant) :: (reste 1 (i + 1))
    else reste (deg_courant + 1) (i + 1)
  in reste 0 0
```

Exercice 5 : Cycle

Soit $n \in \mathbb{N}$. On dit d'une liste l d'entiers deux-à-deux distincts de $[\![0, n-1]\!]$ que c'est une liste *de cycle*. On dit qu'un tableau t' est le permuté d'un tableau t selon une liste de cycle $l = [l_0; l_1; \ldots; l_{p-1}]$ lorsque t' est obtenu à partir de t en déplaçant en case $l_{(i+1) \mod p}$ l'élément se trouvant en case l_i, et ce, pour tout $i \in [\![1, p]\!]$.

Q. 1 Écrire une fonction `permute : int array -> int list -> unit` prenant en paramètres un tableau t de taille n et une liste de cycle l et modifiant le tableau en paramètre de sorte qu'il contienne le permuté du tableau t selon la liste l.

Solution

```
let echange (t: int array) (i: int) (j: int): unit =
  let tmp = t.(i) in
  t.(i) <- t.(j);
  t.(j) <- tmp

let rec permute (t: int array) (l: int list): unit =
  match l with
  | [] | _ :: [] -> ()
  | x :: y :: l' ->
    permute t (y :: l') ;
    echange t x y
```

Q. 2 Définir une fonction `find_opt : (f: int -> bool) (n: int): int option` retournant le plus petit entier `i` de $[\![0, \mathtt{n}-1]\!]$ tel que `(f i)` vaut vrai. Si un tel indice n'existe pas, on retournera `None`.

Solution

```
let find_opt (f: int -> bool) (n: int): int option =
  let rec find_opt_apres (i: int): int option =
    if i = n then None
    else if (f i) then Some i
    else find_opt_apres (i + 1) in
  find_opt_apres 0
```

Q. 3 Écrire une fonction `est_permute : int array -> int array -> int list` prenant en paramètres deux tableaux t et t' de taille n contenant des éléments deux à deux distincts et retournant une liste de cycle l telle que t' est le permuté de t selon la liste l. Si une telle liste n'existe pas, on lèvera un message d'erreur. *Indication : on s'autorisera à utiliser les fonctions du module `List`.*

Solution

```
(* Trouve l'indice de v dans tab, si un tel indice n'existe pas, retourne None. *)
let trouve_indice_de (tab: int array) (v: int): int option =
  find_opt (fun i -> tab.(i) = v) (Array.length tab)

(* Trouve le plus petit indice auquel les tableaux avant et apres
   diffèrent. Si un tel indice n'existe pas, on retourne None. *)
let trouve_indice_different (avant: int array) (apres: int array): int option =
  find_opt (fun i -> avant.(i) = apres.(i)) (Array.length avant)

(* Vérifie que les tableaux avant et apres coïncident, sur les indices en dehors de l *)
exception Non
let rec verifie_differents_dans_liste (avant: int array) (apres: int array) (l: int list):
↪  bool =
```

```
  try
    Array.iteri (fun i x -> if apres.(i) <> x && not (List.mem i l) then raise Non) avant;
    true
  with | Non -> false

let est_cycle (avant: int array) (apres: int array): int list =
  if Array.length avant <> Array.length apres then failwith "Non 1"
  else begin
    match trouve_indice_different avant apres with
    | None            -> []
    | Some(idx_debut) ->
      (* Suit le cycle pour le fabriquer *)
      let rec trouve_cycle (idx: int) (res: int list): int list =
        match trouve_indice_de avant apres.(idx) with
        | None       -> failwith "Non 2"
        | Some(idx') ->
          if idx' = idx_debut then List.rev (res)
          else if List.mem idx' res then failwith "Non 4"
          else trouve_cycle idx' (idx' :: res)
      in
      let ll = trouve_cycle idx_debut [idx_debut] in
      if verifie_differents_dans_liste avant apres ll then ll
      else failwith "Non 3"
  end

```