Exercice 1 : Zipper sur les listes

Un zipper sur une liste est la donnée d'un *curseur* positionné sur un élément. On le représente par le type suivant :

```
type 'a list_zipper = {
  before : 'a list;  (* éléments avant le curseur, en ordre inverse *)
  after  : 'a list;  (* éléments à partir du curseur (tête = élément courant) *)
}
```

Par exemple, le zipper représentant la liste `[1; 2; 3; 4; 5]` avec le curseur positionné sur `3` est : `{ before = [2; 1]; after = [3; 4; 5] }`

Q. 1 Écrire la fonction `of_list : 'a list -> 'a list_zipper option` qui crée un zipper positionné sur le premier élément d'une liste, et renvoie `None` si la liste est vide.

Solution

```
let of_list (l: 'a list) : 'a list_zipper option =
  match l with
  | []   -> None
  | l    -> Some { before = []; after = l }
```

Q. 2 Écrire la fonction `current : 'a list_zipper -> 'a option` qui renvoie l'élément courant, ou `None` si le curseur est en dehors de la liste.

Solution

```
let current (z: 'a list_zipper): 'a option =
  match z.after with
  | []     -> None
  | x :: _ -> Some x
```

Q. 3 Écrire la fonction `to_list : 'a list_zipper -> 'a list` qui reconstruit la liste complète à partir d'un zipper, indépendamment de la position du curseur.

Solution

```
let to_list (z: 'a list_zipper): 'a list =
  List.rev_append z.before z.after
```

Q. 4 Écrire `move_right : 'a list_zipper -> 'a list_zipper option`, qui déplace le curseur d'un cran vers la droite. La fonction renvoie `None` si le curseur est déjà en fin de liste.

Solution

```
let move_right (z: 'a list_zipper): 'a list_zipper option =
  match z.after with
  | _ :: [] | [] -> None   (* fin de liste ou liste vide *)
  | x :: rest    -> Some { before = x :: z.before; after = rest }
```

Q. 5 Écrire `move_left : 'a list_zipper -> 'a list_zipper option`, symétriquement.

Solution

```
let move_left (z: 'a list_zipper) : 'a list_zipper option =
  match z.before with
  | []       -> None
  | x :: rest -> Some { before = rest; after = x :: z.after }
```

Q. 6 Écrire `set : 'a -> 'a list_zipper -> 'a list_zipper option` qui remplace l'élément courant par la valeur donnée. Renvoie `None` si la liste est vide.

Solution

```
let set (v: 'a) (z: 'a list_zipper) : 'a list_zipper option =
  match z.after with
  | []       -> None
  | _ :: rest -> Some { z with after = v :: rest }
```

Q. 7 Écrire `insert_before : 'a -> 'a list_zipper -> 'a list_zipper` qui insère un élément juste avant l'élément courant. Le curseur reste positionné sur l'ancien élément courant.

Solution

```
let insert_before (v: 'a) (z: 'a list_zipper) : 'a list_zipper =
  { z with before = v :: z.before }
```

Q. 8 Écrire `delete : 'a list_zipper -> 'a list_zipper option` qui supprime l'élément courant. Le curseur se positionne alors sur l'élément suivant s'il existe, ou sur le précédent sinon. Renvoie `None` si la liste est vide.

Solution

```
let delete (z: 'a list_zipper): 'a list_zipper option =
  match z.after with
  | [] -> None                          (* liste vide *)
  | [_] ->
    (match z.before with
     | []       -> None                 (* singleton, liste devient vide *)
     | x :: rest -> Some { before = rest; after = [x] })
  | _ :: rest -> Some { z with after = rest }
```

Exercice 2 : Permutations

Dans cet exercice on considère uniquement des tableaux T de taille $n \geqslant 0$, contenant une et une seule fois chaque entier de $[\![0, n-1]\!]$. Étant donné un tableau et un entier $i \in [\![0, n-1]\!]$, on note $o_T(i)$ (on l'appelle l'ordre de i dans le tableau T) le plus petit entier $p > 0$ tel que $\underbrace{T[T[T[\dots[T}_{p \text{ fois}}[i]]]]] = i$.

Q. 1 Définir une fonction `o : int array -> int -> int` prenant en paramètres un tableau T et un entier i et calculant $o_T(i)$.

Solution

```
(** Trouve l'ordre de [i] dans le tableau [t]. *)
let o (t: int array) (i: int): int =
  (** Compte le nombre d'itérations nécessaires avant de revenir sur [i] *)
  let rec retrouve_i (cur: int): int =
    1 + (if t.(cur) = i then 0 else (retrouve_i t.(cur)))
  in (retrouve_i i)
```

Q. 2 Définir une fonction `max_o : int array -> int` prenant en paramètres un tableau T et retournant l'élément d'ordre maximal dans le tableau. On pourra supposer le tableau non vide.

Solution

```
(** Trouve l'ordre maximal dans le tableau [t]. *)
let max_o (t: int array): int =
  let n = Array.length t in
  assert (n <> 0);
  let vu = Array.make n false in

  (** Compte le nombre d'itérations nécessaires avant de revenir sur [i],
      marque les sommets croisés sur le chemin dans [parcourus]. *)
  let rec retrouve_i (cur: int) (i: int): int =
    vu.(i) <- true;
    1 + (if t.(cur) = i then 0 else (retrouve_i t.(cur) i))
  in

  (** Calcule le maximum des ordres des éléments du tableau d'indice >=
      [i] *)
  let rec max_idx_gt_i (i: int): int =
    if i >= n then (-1)
    else if (not vu.(i)) then max (retrouve_i i i) (max_idx_gt_i (i + 1))
    else max_idx_gt_i (i + 1)
  in

  max_idx_gt_i 0
```

Exercice 3 : Matrice creuse

On considère dans cet exercice des matrices carrées d'entiers relatifs : $M \in \mathcal{M}_n(\mathbb{Z})$. Par exemple :

$$\begin{pmatrix} 0 & 3 & 5 \\ 0 & 0 & 7 \\ -1 & 0 & 0 \end{pmatrix}$$

On représente de telles matrices de deux manières :

- Ou bien par un tableau de tableau d'entiers relatifs, par exemple : `[|[|0; 3; 5|]; [|0; 0; 7|]; [|-1; 0; 0|]|]`.
- Ou bien par une liste de triplets : (i, j, v) indiquant que dans la case d'indice (i, j) de la matrice se trouve la valeur v. Si une case n'est pas mentionnée par une telle liste, c'est que la matrice représentée contient 0. Par exemple pour la matrice ci-dessus `[(0, 1, 3); (0, 2, 5); (1, 2, 7); (2, 0, -1)]`.

Q. 1 Donner des fonctions de conversion d'une représentation dans l'autre.

Solution

```
type m1 = int array array
type m2 = (int * int * int) list

(* Retourne les dimensions de la matrice m. *)
let max_coord (m: m2): int * int =
  List.fold_left (fun (max_i, max_j) (i, j, v) ->
      (max max_i i, max max_j j)
    ) (0, 0) m

let m1_of_m2 (m: m2): m1 =
  let dimx, dimy = max_coord m in
  let res = Array.make_matrix dimx dimy 0 in
  List.iter (fun (i, j, v) ->
      res.(i).(j) <- v
    ) m;
  res

let m2_of_m1 (m: m1): m2 =
  let res = ref [] in
  Array.iteri (fun i l ->
      Array.iteri (fun j v ->
          if v <> 0 then res := (i, j, v) :: !res
        ) l
    ) m;
  !res
```

Exercice 4 : Découpage en deux de listes en C

On dispose (dans le fichier joint) d'une définition d'un type liste en C, que l'on rappelle ci-dessous.

```c
struct list_s {
  int elem ;                    /* L'élément en tête de liste */
  struct list_s* next ;         /* Le suivant */
};

typedef struct list_s* liste ;
```

Q. 1 Écrire une fonction C `void decoupe_en_deux(liste entree, liste* r1, liste* r2)` qui partage la liste `entree` en deux moitiés en place, sans allouer de nouveaux nœuds ni copier de valeurs. Après l'exécution `*r1` devra donc pointer vers le début d'une liste contenant les premiers éléments de `entree`, `*r1` devra pointer vers le début d'une liste contenant les derniers éléments de `entree`, Si la liste a un nombre impair d'éléments, la première moitié est la plus courte.

Indication : Utiliser deux curseurs, l'un avançant d'un nœud à chaque étape, l'autre de deux. Quand le curseur rapide atteint la fin de la liste, le curseur lent se trouve au milieu.

Solution

```c
void decoupe_en_deux(liste entree, liste* r1, liste* r2) {
  liste avant_lent = NULL;
  liste lent       = entree ;
  liste rapide     = entree ;
  while (rapide != NULL && rapide->next != NULL) {
    avant_lent = lent ;
    lent       = lent->next ;
    rapide     = rapide->next->next;
  }
  *r1 = entree ;
  if (avant_lent != NULL) {     /* Au moins deux éléments dans la liste */
    avant_lent->next = NULL;
    *r2 = lent ;
  } else {                      /* Moins de deux éléments */
    *r2 = NULL ;
  }
}
```

Q. 2 Écrire la fonction C `void decoupe_en_deux(liste entree, liste* r1, liste* r2)` qui partage la liste `entree` en deux moitiés en place, sans allouer de nouveaux nœuds ni copier de valeurs. Les nœuds doivent être répartis selon leur position : les nœuds aux positions paires forment `*r1`, ceux aux positions impaires forment `*r2`. L'ordre relatif des nœuds est conservé dans chaque sous-liste. Aucun nœud n'est alloué ou copié.

Solution

```c
void decoupe_en_deux_bis(liste entree, liste* r1, liste* r2) {
  *r1 = entree ;
  if (entree == NULL) {
    *r2 = NULL ;
    return ;
  } else {
    *r2 = entree->next ;
    liste curseur = entree ;
    while (curseur -> next != NULL) {
```

```
10        liste apres = curseur -> next ;
11        curseur->next = curseur->next->next ;
12        curseur = apres ;
13      }
14      return ;
15    }
16  }
```