TD corrigé listes doublement chaînées
Option informatique

On rappelle qu'une **structure de donnée** est un moyen de stocker un ensemble d'éléments. On distingue:

- Structure **persistante**: ne peut pas être modifiée, seules de nouvelles valeurs sont renvoyées.
  Exemple: `list`, `string`, `arbre`...
- Structure **impérative** (aussi appelé **mutable**): peut être modifiée.
  Exemple: `array`, tous les types contenant `ref` ou `mutable`...

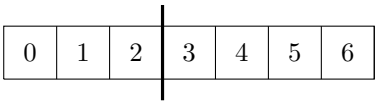
Le type `list` d'OCaml ne permet de parcourir les éléments que dans un seul sens (on ne peut pas parcourir une liste « à l'envers »). Il existe une structure de donnée classique appelée **liste doublement chaînée** qui permet de le faire. Nous allons en voir deux implémentations possible.

I Implémentation persistante d'une liste doublement chaînée: le zipper

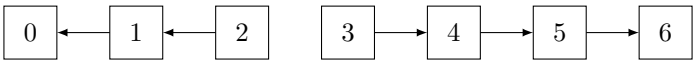
Un zipper est une structure persistante permettant de se déplacer à droite ou à gauche dans une liste. La position en cours est appelée curseur. Pour implémenter un zipper, on utilise deux `list` (les éléments avant le curseur sont dans `left`, ceux après dans `right`):

```
type 'a zipper = { left : 'a list; right : 'a list };;
```

Par exemple, soit le zipper suivant (dont le curseur est entre 2 et 3):



Ce zipper est représenté par { `left = [2; 1; 0]; right = [3; 4; 5; 6]` }:



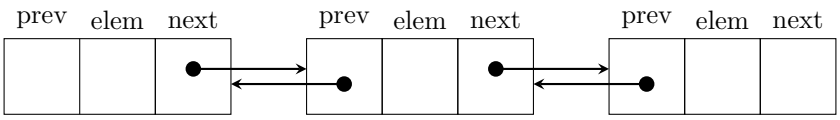
1. Écrire une fonction `move_right : 'a zipper -> 'a zipper` déplaçant le curseur d'un zipper vers la droite (on renverra une erreur si ce n'est pas possible). De façon similaire on pourrait le déplacer à gauche.
2. Écrire des fonctions pour supprimer et ajouter un élément juste à droite du curseur d'un zipper, en O(1).
3. Écrire une fonction pour convertir un `zipper` en `list`.

▶ Voici les fonctions demandées:

```
let move_right z = match z.right with
  | [] -> failwith "empty right"
  | e::q -> { left = e::z.left; right = q };;

let add e z = { left = z.left; right = e::z.right };;

let del z = match z.right with
  | [] -> failwith "empty right"
  | e::q -> { left = z.left; right = q };;

let list_of_zipper z = (List.rev z.left) @ z.right;;
```

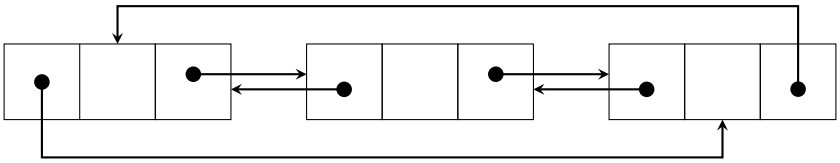
II Liste doublement chaînée cyclique impérative

On pourrait définir une liste doublement chaînée en utilisant un type contenant un prédécesseur `prev`, un successeur `next` et un élément `elem`, ce qui ressemblerait à ceci:



Cependant il faudrait aussi gérer les extrémités, ce qui est un peu compliqué (il faut traiter le cas où il n'y a pas de prédécesseur ou successeur).
On va utiliser à la place une liste doublement chaînée **cyclique**, où le prédécesseur du 1er élément est le dernier élément:

```
type 'a l2c = {elem : 'a; mutable prev : 'a l2c; mutable next : 'a l2c};;
```



Pour créer une nouvelle `l2c` à 1 élément, il faut que celui-ci soit son propre successeur et prédécesseur, ce que permet aussi `rec` (on appelle ceci une variable récursive):

```
let create e = (* renvoie une l2c contenant seulement e *)
  let rec l = {elem = e; prev = l; next = l} in
  l;;
```

1. Écrire des fonctions `add`/`del` pour ajouter/supprimer un élément dans une `l2c` en O(1).
   ▶

```
let add l e = (* add after l *)
  let l_new = {elem = e; prev = l; next = l.next} in
  l.next.prev <- l_new;
  l.next <- l_new;;
let del l = (* del l *)
  l.prev.next <- l.next;
  l.next.prev <- l.prev;;
```

2. Écrire des fonctions `length` et `mem` pour calculer la taille d'une `l2c` et pour tester si un élément appartient à une `l2c`. Pour savoir quand arrêter de parcourir la `l2c`, on utilisera `==` ou `!=` qui teste si deux objets sont stockés au même endroit en **mémoire** (à ne pas confondre avec `=` et `<>` qui testent l'égalité en **valeur**). Complexité?
   ▶ On peut utiliser une fonction récursive ou un `while`, en regardant les `next` jusqu'à revenir sur l'élément initial. Les 2 fonctions parcourent chaque élément en effectuant un nombre constant d'opérations, d'où une complexité O(taille de la liste `l`).

```
let length l =
  let rec aux l1 =
    if l1 == l then 1 else 1 + aux l1.next in
  aux l.next;;
let mem e l =
  let cur = ref l.next in
  while !cur.elem <> e && !cur != l do
    cur := !cur.next
  done;
  !cur.elem = e;;
```

3. Écrire une fonction `fusion` pour réaliser l'union de deux `l2c`.
   Comment aurait-on pu utiliser `fusion` pour implémenter `add`?
   ▶ On aurait pu écrire `add` en utilisant `fusion`: `let add l e = fusion l (create e);;`

```
let fusion l1 l2 =
  (l1.next).prev <- l2;
  (l2.next).prev <- l1;
  let l1n = l1.next in
  l1.next <- l2.next;
  l2.next <- l1n;;
```