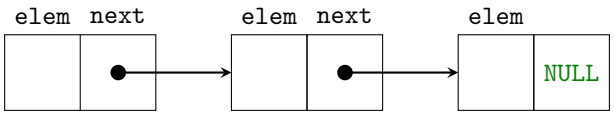
On considère un type `linked_list` de liste simplement chaînée impérative (chaque élément a accès à l'élément suivant `next`) :

```
type 'a cell = { elem : 'a, next : 'a linked_list }
and 'a linked_list = E | C of 'a cell
```



Ici on utilise `and` pour définir 2 types simultanéments (de la même façon que l'on peut définir 2 fonctions récursives mutuellement dépendantes avec `and`). On a besoin de le faire afin de gérer la fin de liste (Vide).

1. Écrire une/des instruction(s) OCaml pour définir une liste simplement chaînée `l` contenant les entiers 1 et 2.

2. Écrire une fonction `to_list : 'a linked_list -> 'a list` convertissant une liste simplement chaînée en `list` classique.

Il est possible qu'une liste simplement chaînée possède un cycle, si l'on revient sur le même élément après avoir parcouru plusieurs successeurs. Dans ce cas, la fonction `to_list` précédente ne termine pas...
On souhaite donc déterminer algorithmiquement si une liste simplement chaînée `l` possède un cycle.

3. Écrire une fonction récursive `has_cycle : 'a linked_list -> 'a list -> bool` telle que `has_cycle l vus` détermine si `l` possède un cycle, en stockant les éléments déjà rencontrés dans `vus` (initialement on utilise une liste vide pour `vus`).
   Si n est le nombre d'éléments de `l`, quelle est la complexité de cet algorithme, en temps et en mémoire?

Il existe un algorithme plus efficace, appelé algorithme du lièvre et de la tortue (ou: algorithme de Floyd/algorithme rho de Pollard, très utile en cryptographie).
Il consiste à initialiser une variable `tortue` à la case de `l`, une variable `lievre` à la case suivante, puis, tant que c'est possible:

- Si `lievre` et `tortue` font référence à la même case, affirmer que `l` contient un cycle.
- Sinon, avancer `lievre` de deux cases et `tortue` d'une case.

4. Montrer que cet algorithme permet bien de détecter un cycle. Quelle est sa complexité en temps et en espace?

5. Comment obtenir la longueur du cycle, s'il existe?

6. Écrire une fonction utilitaire `step : 'a linked_list -> 'a linked_list` telle que `step l` avance `l` d'une case ou renvoie `Vide` si `l = Vide`.

7. Écrire une fonction récursive `has_cycle : 'a linked_list -> 'a linked_list -> bool` implémentant l'algorithme du lièvre et de la tortue, dont les deux arguments sont les positions actuelles du lièvre et de la tortue (pour savoir si `l` contient un cycle, on appelera donc `has_cycle (step l) l`).
   On pourra utiliser == qui compare 2 objets en **mémoire** (à ne pas confondre avec = qui les compare en **valeur**).