

## TP D'OPTION INFORMATIQUE 1

### Manipulation de listes en Caml

L'objectif de ce TP est d'écrire les fonctions élémentaires sur les listes. On commencera systématiquement par déterminer le type de la fonction ou de l'objet à définir. On s'interdira bien entendu d'utiliser les implémentations déjà existantes en Caml. Chaque fonction devra être testée.

1. Écrire une fonction `longueur` prenant en argument une liste et renvoyant sa longueur (cette fonction existe déjà sous le nom `List.length`, et on notera bien qu'elle a un coût linéaire).
2. Écrire une fonction `tete` renvoyant le premier élément d'une liste non vide (cette fonction existe déjà sous le nom de `List.hd`).  
On pourra utiliser `failwith` suivi d'une chaîne de caractère pour déclencher une erreur dans le cas où la liste est vide.
3. Écrire une fonction `queue` renvoyant la liste de tous les éléments d'une liste non vide, sauf le premier (cette fonction existe déjà sous le nom de `List.tl`).
4. Écrire une fonction `appartient` prenant en argument un élément et une liste et testant si cet élément appartient à la liste (cette fonction existe déjà sous le nom `List.mem`).
5. Écrire une fonction `nombre_occurrences` prenant en argument un élément  $x$  et une liste  $l$  et renvoyant le nombre d'occurrences de  $x$  dans  $l$ .
6. Écrire une fonction `images` prenant en argument une fonction  $f$  et une liste  $l$  et renvoyant la liste des images par  $f$  des éléments de  $l$  (cette fonction existe déjà sous le nom de `List.map`).
7. Écrire une procédure `iterer` prenant en argument une procédure  $p$  et une liste  $l$  et appliquant  $p$  sur chaque élément de  $l$  (cette fonction existe déjà sous le nom de `List.iter`).  
On peut enchaîner deux instructions en Caml en les séparant par ;
8. Écrire une fonction `dernier` renvoyant le dernier élément d'une liste non vide.
9. Écrire une fonction `maximum` renvoyant le maximum d'une liste non vide.
10. Écrire une fonction `indice` prenant en argument un élément  $a$  et une liste  $l$  et renvoyant un indice, s'il existe, où apparaît  $a$  dans  $l$ .
11. Écrire une fonction `nieme` prenant en argument une liste  $l$  et un entier  $n$  et une liste  $l$  et renvoyant l'élément d'indice  $n$  (cette fonction existe déjà sous le nom `List.nth`. On notera bien qu'elle a un coût linéaire en  $n$ ).
12. Écrire une fonction `valeur_associee` prenant en argument un élément  $c$  et une liste de couples  $l$ , et renvoyant un élément  $v$  tel que  $(c, v)$  apparaît dans  $l$ , si un tel  $v$  existe (cette fonction existe déjà sous le nom `List.assoc`).
13. Écrire une fonction `concatener` renvoyant la concaténation de deux listes (cette fonction existe déjà sous le nom de `List.append`, attention à la différence avec Python! La concaténation peut également s'écrire avec l'opération infix `@`. On notera bien qu'elle a un coût linéaire en la longueur de la première liste).
14. Écrire une fonction `aplatissement` prenant en argument une liste de listes et les concaténant (cette fonction existe déjà sous le nom `List.flatten`).
15. Écrire une fonction `pour_tout` prenant en argument un prédicat  $p$  (c'est-à-dire une fonction de type de retour booléen) et une liste  $l$  et qui teste si tous les éléments de  $l$  vérifient  $p$  (cette fonction existe déjà sous le nom `List.for_all`).

16. Écrire une fonction `existe` prenant en argument un prédicat  $p$  et une liste  $l$  et qui teste s'il existe un élément de  $l$  vérifiant  $p$  (cette fonction existe déjà sous le nom `List.exists`).
17. Écrire une fonction `filtre` prenant en argument un prédicat  $p$  et une liste  $l$  et renvoyant la liste des éléments de  $l$  vérifiant  $p$  (cette fonction existe déjà sous le nom `List.filter`).
18. Écrire une fonction `trouver` prenant en argument un prédicat  $p$  et une liste  $l$  et renvoyant le premier élément de  $l$  vérifiant  $p$ , s'il existe (cette fonction existe déjà sous le nom `List.find`).
19. Écrire une fonction `partition` prenant en argument un prédicat  $p$  et une liste  $l$  et renvoyant le couple formé par la liste des éléments de  $l$  vérifiant  $p$  et la liste des éléments de  $l$  ne vérifiant pas  $p$  (cette fonction existe déjà sous le nom `List.partition`).
20. Écrire une fonction `combinaison` prenant en argument deux listes supposées de même longueur, et renvoyant la liste des couples termes à termes de ces deux listes (cette fonction existe déjà sous le nom `List.combine`).
21. Écrire une fonction `separation` prenant en argument une liste de couples et renvoyant le couple de listes correspondant (cette fonction existe déjà sous le nom `List.split`).
22. Écrire une fonction `images_iterees` prenant en argument un élément  $a$ , une fonction  $f$  et un entier  $n$ , et renvoyant la liste  $[a, (f a), (f (f a)), \dots, (f^n a)]$ .
23. Définir la liste des entiers de 0 à 1000, dans l'ordre décroissant.
24. Définir la liste des entiers de 0 à 1000, dans l'ordre croissant.
25. Écrire une fonction `parcours_droite` prenant en argument une fonction  $f$ , une liste  $l = [a_1; \dots; a_n]$  et un élément  $b$ , et renvoyant  $f a_1 (f a_2 (\dots (f a_n b) \dots))$  (cette fonction existe déjà sous le nom `List.fold_right`).
26. Retrouver la fonction `somme`, prenant une liste d'entiers et renvoyant la somme de ses éléments, comme une application de la fonction précédente. On pourra utiliser `(+)` l'opérateur d'addition vu comme une fonction `int -> int -> int`.
27. On souhaite écrire une fonction `renverser` prenant en argument une liste  $l$  et renvoyant la liste de sens contraire (cette fonction existe déjà sous le nom `List.rev`). Une version naïve pourrait s'écrire :

```
let rec renverser l =
  match l with
  [] -> []
  | a::q -> (renverser q)@[a]
```

Cette version a le défaut réhhibitoire d'être de complexité quadratique, car la complexité de la concaténation `l1@l2` est linéaire en la longueur de `l1`. Écrire une version de `renverser` de complexité linéaire. On utilisera une fonction auxiliaire prenant en argument une liste  $l2$  et une liste  $r$ , supposant que  $l = r'@l2$ , où  $r'$  est le renversé de  $r$ , et renvoyant le renversé de  $l$ .
28. Écrire une fonction `parcours_gauche` prenant en argument une fonction  $f$ , un élément  $a$  et une liste  $l = [b_1; \dots; b_n]$ , et renvoyant  $f (\dots (f (f a b_1) b_2) \dots) b_n$  (cette fonction existe déjà sous le nom `List.fold_left`).
29. Retrouver `renverser` comme une application de la fonction précédente.
30. Écrire une fonction `produit_cartesien` prenant en argument deux listes  $l1$  et  $l2$  et renvoyant une liste contenant tous les couples  $(a, b)$  où  $a$  est un élément de  $l1$  et  $b$  un élément de  $l2$ .