Listes

On rappelle la définition des listes polymorphes :

```
Inductive liste (A:Set) : Set :=
| nil  : liste A
| cons : A -> liste A -> liste A.
```

Lisez le principe d'induction `liste_ind` produit par cette définition.

Map

1. Écrivez la fonction `length` qui calcule la longueur d'une liste.
2. Écrivez la fonction `map` qui applique une fonction f (de A vers B) à chaque élément d'une liste.
3. Démontrez que la longueur du résultat de `map` est la longueur de la liste originale.
4. Définissez une fonction `compose` qui produit la composition $f \circ g$ de deux fonctions f et g passées en paramètre.
5. Démontrez que la composition est associative, c'est-à-dire que $f \circ (g \circ h) = (f \circ g) \circ h$. (Indication : `unfold`.)
6. Démontrez que, ∀`(A B C: Set)`, ∀`(f: B->C)`, ∀`(g: A->B)`, ∀`(l: liste A)`,

$$\texttt{map}\ (f \circ g)\ l = \texttt{map}\ f\ (\texttt{map}\ g\ l)$$

Paramètres implicites

7. Utilisez la commande `Arguments` pour définir comme implicites tous les paramètres de type `Set` de `nil`, `cons`, `length`, `map` et `compose`. Vous ferez de même pour toutes les fonctions à définir ci-dessous.

Append et reverse

8. Écrivez une fonction `append` qui concatène deux listes. (Indication : évitez la récursivité terminale.)
9. Démontrez que `map f (append l1 l2) = append (map f l1) (map f l2)`.
10. Écrivez une fonction `reverse` qui inverse l'ordre des éléments d'une liste. (Indication : en utilisant `append`.)
11. Démontrez que `map f (reverse l) = reverse (map f l)`.
12. Prouvez que `reverse (append l1 l2) = append (reverse l2) (reverse l1)`.
13. Démontrez que `reverse (reverse l) = l`.
14. Formulez et démontrez deux lemmes qui expliquent l'effet de `append` et `reverse` sur la longueur des listes. (Indication : importez `Arith` et utilisez `ring` pour l'égalité finale, ou simplement le lemme `plus_comm`.)

Fold right (et left)

15. Écrivez une fonction `foldr` qui, à partir d'une fonction f, d'une valeur z, et d'une liste $[x_1, x_2, \ldots, x_n]$, calcule :

$$f(x_1, f(x_2, f(\cdots, f(x_n, z)\cdots)))$$

16. Démontrez que `foldr f z (append l1 l2) = foldr f (foldr f z l2) l1`.
17. Démontrez que `foldr (fun h q => cons (f h) q) nil l = map f l`.
18. De façon similaire, démontrez que `foldr (fun ...) 0 l = length l` après avoir complété le premier paramètre de `foldr`.
19. Même question pour `foldr (fun ...) nil l = l`.
20. Même question pour `foldr (fun ...) nil l = reverse l`.
21. Écrivez une fonction `foldl`, qui prend les mêmes paramètres que `foldr`, et calcule $f(\cdots f(f(z, x_1), x_2)\cdots, x_n)$.
22. Démontrez que `foldr f z (reverse l) = foldl (flip f) z l`, où `flip f` représente la fonction `fun x y => f y x`. (Indication : utilisez `generalize z` puis `clear z` après `intros` pour obtenir une hypothèse d'induction plus générale.)

Zip et unzip

23. Définissez une fonction `zip` qui prend en paramètre deux listes et construit une liste de couples. Par exemple :

    `zip (cons 1 (cons 2 (cons 3 nil))) (cons 11 (cons 12 nil))`

    produit la liste `cons (1,11) (cons (2,12) nil)`. Notez que le résultat n'est pas plus long que la plus courte des deux listes.
24. Démontrez que `length (zip l1 l2) = length (zip l2 l1)`.
25. Démontrez que, pour toute liste `l1` et `l2` :

    `length (zip l1 l2) = length l1 \/ length (zip l1 l2) = length l2`
26. Démontrez que :

    `zip (map f1 l1) (map f2 l2) = map (prodf f1 f2) (zip l1 l2)`

    après avoir donné une définition appropriée de `prodf`. (Indication : `fst` (resp. `snd`) extraie le premier (resp. second) élément d'un couple. Vous pouvez aussi utiliser directement `match p with (x, y) => x end` au lieu de `fst`.)
27. Définissez la fonction `unzip`, de type `liste (A*B) -> liste A * liste B`, qui scinde une liste de couples en un couple de listes. (Indication : pour obtenir les éléments du résultat d'un appel récursif, utilisez :
    - soit `match (unzip q) with (ta,tb) => ... end`.
    - soit `let (ta,tb) := unzip q in ...`, qui est exactement équivalent.)
28. Démontrez que `length (fst (unzip l)) = length l`, puis la même chose avec `snd` si vous voulez. (Indication : pour décomposer le terme `(unzip l)`, utilisez `case_eq (unzip l)`, suivi de `intros`.)
29. Démontrez que `zip` appliqué aux résultats de `unzip` produit la liste initiale. (Indication : `injection` transforme une égalité entre couples en deux égalités.)