

Bases sur les listes

Toutes les fonctions qui suivent doivent être définies récursivement.

- 1) Écrire la fonction `longueur : 'a list -> int` renvoyant la taille d'une liste.
- 2) Écrire la fonction `appartient : 'a list -> 'a -> bool` permettant de savoir si un élément donné appartient à une liste.
- 3) Écrire la fonction `nb_occ : 'a list -> 'a -> int` donnant le nombre d'occurrences d'un élément dans une liste.
- 4) Écrire la fonction `somme : int list -> int` calculant la somme des éléments d'une liste.
- 5) Supprimer les occurrences d'un élément dans une liste, par exemple supprimer tous les 1 d'une liste d'entiers. Type : `supprime : 'a -> 'a list -> 'a list`
- 6) On veut retourner une liste, c'est-à-dire transformer la liste $[a_1; a_2; \dots; a_n]$ en la liste $[a_n; a_{n-1}; \dots; a_1]$.
Écrire une fonction auxiliaire `ajoute_fin : 'a -> 'a list -> 'a list` qui ajoute un élément x à la fin d'une liste donnée.
Programmer ensuite, en utilisant la fonction auxiliaire, la fonction `retourne : 'a list -> 'a list`
Quelle est la complexité de cette fonction ?
Programmer une meilleure version en utilisant un accumulateur (initialement vide)
Type : `retourne_bis : 'a list -> 'a list -> 'a list`
- 7) Rechercher le minimum d'une liste supposée non vide. Type : `minimum : 'a list -> 'a`
La fonction n'utilisera ni accumulateur, ni fonction auxiliaire.
- 8) Concaténer deux listes : la fonction `concat : 'a list -> 'a list -> 'a list` devra être telle que si on lui passe les listes

$$[a_0; \dots; a_{n-1}] \text{ et } [b_0; \dots; b_{m-1}]$$

elle renvoie la liste

$$[a_0; \dots; a_{n-1}; b_0; \dots; b_{m-1}].$$

Bases sur les tableaux

Toutes les fonctions qui suivent ne doivent pas être définies récursivement. De plus, vous devez, quand cela est possible, renvoyer la réponse sans avoir terminé le parcours du tableau (pour cela, il faut utiliser une boucle `while`).

- 9) Écrire la fonction `longueur : 'a array -> int` renvoyant la taille d'un tableau.
- 10) Écrire la fonction `appartient : 'a array -> 'a -> bool` permettant de savoir si un élément donné appartient à un tableau.
- 11) Écrire la fonction `nb_occ : 'a array -> 'a -> int` donnant le nombre d'occurrences d'un élément dans un tableau.
- 12) Écrire la fonction `somme : int array -> int` calculant la somme des éléments d'un tableau.

- 13) Supprimer les occurrences d'un élément dans un tableau, par exemple supprimer tous les 1 d'un tableau d'entiers. Type : `supprime : 'a -> 'a array -> 'a array` Le tableau lui même n'est pas modifié, on renvoie un nouveau tableau qui sera plus petit.
- 14) On veut retourner un tableau, c'est-à-dire transformer le tableau $[a_1; a_2; \dots; a_n]$ en le tableau $[a_n; a_{n-1}; \dots; a_1]$.
Type : `retourne : 'a array -> unit`
- 15) Rechercher la valeur minimale dans un tableau supposé non vide. Type : `minimum : 'a array -> 'a`

Exercices sur les listes

- 16) Scission d'une liste en deux listes.
On part d'une liste

$$[a_0; \dots; a_{n-1}]$$

et on la coupe en deux pour obtenir

$$[a_0; a_2; a_4; \dots] \text{ et } [a_1; a_3; a_5; \dots]$$

Type : `scinde : 'a list -> 'a list * 'a list`

- 17) On crée un type enregistrement `suite_somme` par

```
type suite_somme = {suite : int list ; somme : int}
```

Le but est que la rubrique `somme` contienne la somme des éléments de la liste `suite`.

- Écrire une fonction `est_valide : suite_somme -> bool` qui teste pour savoir si la rubrique `somme` contient bien la somme des termes de la rubrique `suite`.
 - Écrire une fonction `etete : suite_somme -> suite_somme` qui enlève le premier élément de la liste (et modifie la somme). La fonction lèvera une exception si la liste est vide.
 - Écrire une fonction `add : int -> suite_somme -> suite_somme` qui ajoute un élément au début de la liste (et modifie la somme).
 - Écrire une fonction `creation : int -> int -> suite_somme` tel que `creation n p` renvoie l'élément dont la liste est composée de `n` fois l'entier `p`.
- 18) Fonction `list_it` : On donne une fonction `f`, une liste `l = [a1; a2; ...; an]` et une valeur initiale `b`. L'instruction `list_it f l b` calcule successivement `b, f an b` puis `f an-1 (f an b)` etc... La valeur de `b` est utilisée comme argument du premier appel de `f` l'autre argument étant `an`, puis le résultat est placé comme argument du second appel pour `f` l'autre argument étant `an-1`, etc...
- Ecrire la fonction `list_it` de type

```
val list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```
 - Utiliser la fonction `list_it` pour calculer la somme des éléments d'une liste.
 - Utiliser la fonction `list_it` pour concaténer deux listes.

- 19) Éliminer les doublons d'une liste : programmer une fonction `elimine_doublons` prenant une liste en paramètre et renvoyant une copie de cette liste sans répétition d'un même élément. On conservera l'élément lors de sa première apparition. Par exemple : la liste `[1; 7; 5; 7; 13; 5]` deviendra `[1; 7; 5; 13]`.
- 20) Suite de Thue-Morse.
- a) Définir une fonction `base_list` de type `base_list : int list -> int list` qui prend en argument une liste de 0 et de 1 et qui remplace :
- chaque 0 par les éléments 0 et 1
 - chaque 1 par les éléments 1 et 0.
- Par exemple, `base_list [0;0;1] = [0;1;0;1;1;0]`.
- b) Définir une fonction `thue_Morse` de type `thue_Morse : int -> int list -> int list` telle que `thue_Morse n l` renvoie la liste obtenue à partir de `l` en appliquant n fois la fonction `base_list`.
- 21) Écrire une fonction récursive `liste_premiers : int -> int list` permettant d'obtenir la liste des nombres premiers inférieurs ou égaux à un entier `n`.