Programmation fonctionnelle 2025-2026

Feuille 7 : Listes

Exercice 7.1 1. Réécrire avec le type prédéfini `list` de `OCaML`, les fonctions `interval_list`, `map`, `list_length` et `filter` écrites précédemment avec le type `'a mylist` .

2. Écrire les fonctions de conversion entre les listes de type `'a mylist` et les listes prédéfinies en `OCaML` (`list_of_mylist`, `mylist_of_list`).

Exercice 7.2 1. Écrire une fonction `replicate x k` qui construit la liste composée de `k` répétitions de l'élément `x` .

2. Quel est le type de votre fonction ?

3. Écrire une version récursive terminale de cette fonction.

Exercice 7.3 1. Écrire une fonction `reverse l` qui prend en paramètre une liste `l` quelconque et retourne une liste constituée des éléments de `l` en sens inverse.

2. Indiquer si la fonction écrite est récursive terminale. Donner sa complexité.

3. Donner une version linéaire et récursive terminale.

Exercice 7.4 1. Écrire une fonction `iota_r` qui prend en argument un entier `n` et renvoie la liste `[n; n-1; ...; 1]` .

2. Écrire une fonction `iota` qui prend en argument un entier `n` et renvoie la liste `[1; ...; n-1; n]` .

3. Donner les complexités de vos fonctions.

4. Donner des versions linéaires et récursives terminales de ces fonctions.

5. Tester les fonctions avec des valeurs de `n` de plus en plus grandes.

6. Expliquer pourquoi la solution suivante est mauvaise :

```
let rec bad_iota n =
  if n = 0
  then []
  else (bad_iota (n - 1)) @ [n]
```

Exercice 7.5 1. Écrire une fonction `member` qui teste si son premier argument `x` appartient à la liste donnée par son second argument `l` . Cette fonction renvoie donc soit `true` , soit `false` .

2. Quel est le type de cette fonction ?

3. La fonction demandée existe dans la bibliothèque standard `OCaML` et s'appelle `List.mem` . Comparer votre code à celui de la fonction `List.mem` .

Exercice 7.6 1. Écrire une fonction `count x l` qui compte le nombre de fois que son premier argument `x` appartient à la liste donnée par son second argument `l` . Cette fonction renvoie donc un entier.

2. Quel est le type de cette fonction ? Comparer ce type avec celui de la fonction `member` .

3. Réécrire la fonction `member` en utilisant `count` . On appellera cette nouvelle fonction `member'` .

Exercice 7.7 Un moyen de tester la différence d'efficacité entre deux fonctions est d'utiliser la commande unix `time`, qui renvoie le temps CPU, en secondes, utilisé par le programme depuis le début de son exécution. `OCaml` permet en fait de le faire dans le programme lui-même : il suffit d'enregistrer ce temps avant et après l'évaluation d'un bloc de code et de faire une soustraction pour connaître le temps CPU pris par ce bloc.

Le code suivant permet de le faire pour les fonctions `member` et `List.mem`.

```
let l = iota 100000

let time f =
  let start = Sys.time() in
  let _ = f () in
  Sys.time() -. start

# time (fun () -> member 99999 l);;
- : float = 3.0000000001972893e-06
# time (fun () -> List.mem 99999 l);;
- : float = 2.99999999997524469e-06
```

Tester la différence d'efficacité des fonctions `member` et `member'`.

Exercice 7.8 1. Utiliser la fonction `List.mem` pour écrire une fonction `list_subset` qui prend en argument deux listes `l` et `l'`, et qui teste si tout élément de `l` apparaît dans `l'`.

2. Tester la fonction `list_subset` sur plusieurs exemples bien choisis.

Exercice 7.9 Une liste `l` est une *permutation* d'une liste `l'` si elle est composée des mêmes éléments, chacun apparaissant dans les deux listes avec le même nombre d'occurrences, mais éventuellement dans un autre ordre. Par exemple, `[1;2;3;3]` est une permutation de `[3;2;3;1]`, mais pas de `[1;2;3]`, car 3 apparaît un nombre de fois différent dans les 2 listes.

1. Utiliser la fonction `count` vue précédemment pour écrire une fonction qui prend en argument deux listes `l` et `l'`, et qui teste si `l` est une permutation de `l'`.
2. Tester la fonction sur plusieurs exemples bien choisis.

Exercice 7.10 Une liste `l` est un *prefixe* d'une liste `l'` s'il existe une liste `l''` telle que `l'= l @ l''`.

1. Écrire une fonction qui prend en arguments deux listes `l` et `l'` et qui teste si `l` est un préfixe de `l'`.
2. Tester !

Exercice 7.11 1. Écrire une fonction récursive `squares` qui prend un argument une liste d'entiers et qui renvoie la liste des carrés de ces entiers.

2. Réécrire la fonction `map` en récursif terminal.
3. Réécrire la fonction `squares` en utilisant la fonction `map`.

Exercice 7.12 1. Écrire une fonction `sum` qui prend un argument une liste d'entiers et qui calcule la somme de ses éléments.

2. Écrire une fonction `prod` qui prend un argument une liste d'entiers et qui calcule le produit de ses éléments.

3. Tester la dernière fonction sur la liste `0 :: (iota 10000)`.
4. Améliorer la fonction pour le cas où la liste contient un élément nul.

Exercice 7.13 — Fonction break. Soit la fonction `break` suivante :

```
let rec break pred l =
  match l with
    [] -> [], []
  | e :: t -> let l1, l2 = break pred t
    in if pred e then e :: l1, l2 else l1, e :: l2
```

1. Quel est le type de la fonction `break` ?
2. Quelle est la valeur, en général, de `break p l` ?
3. Écrire une version récursive terminale `break_rt` de la fonction `break`. Si vous utilisez une fonction auxiliaire, expliquez ce qu'elle calcule en fonction de ses arguments.