Manipulations de listes // Éléments de correction

I. Échauffement

Question 1

- ⋆ Attention à considérer TOUS les cas, y compris les listes de longueur 0 ou 1.
- ⋆ Il faut ABSOLUMENT éviter d'utiliser `List.length` dont le coût est linéaire en la longueur de la liste.
- ⋆ Le filtre `[ a; _ ]` peut aussi s'écrire `a :: [ _ ]` ou encore `a :: _ :: []`

```
let rec avant_dernier l =
  match l with
  | [] -> failwith "avant_dernier"
  | [ _ ] -> failwith "avant_dernier"
  | [ a; _ ] -> a
  | _ :: l -> avant_dernier l
;;
```

On peut remarquer que le filtre `[ _ ]` est inutile. Pourquoi ?

Question 2 C'est l'occasion de rappeler l'usage des parenthèses en OCaml. Elles servent UNIQUEMENT à regrouper des expressions en cas d'ambigüité. Écrire `f(a)` ne sert à rien, alors que `(f a)` peut être utile.

```
let rec appli f l =
  match l with
  | [] -> []
  | a :: l' -> f a :: appli f l'
;;
```

II. Quelques fonctions supplémentaires

Question 3

```
let rec longueur l =
  match l with
  | [] -> 0
  | _ :: l' -> 1 + longueur l'
;;
```

Question 4 La somme des éléments de la liste vide est nulle. Une autre façon de voir cela est que 0 est l'élément neutre de l'addition.

```
let rec somme l =
  match l with
  | [] -> 0
  | a :: l' -> a + somme l'
;;
```

Question 5 Cette fonction est un peu délicate. Il y a beaucoup de propositions avec un type cohérent, mais qui sont fausses. Essayez votre version. De toutes façons, je rappelle que vous devez toujours essayer vos fonctions sur, au moins, des exemples simples.

```
let rec suffixes l =
  match l with
  | [] -> [ [] ]
  | a :: l' -> l :: suffixes l'
;;
```

Question 6

- ⋆ Remarquons que les listes peuvent être vides. Dans ce cas, le `pour_tous` doit renvoyer `false` (car on ne peut pas trouver l'élément ne vérifiant pas `prop`), et `true` pour `il_existe`.
- ⋆ J'utilise ici la forme compacte du test, en profitant de l'aspect paresseux de l'évaluation. Ainsi, à la place de `if b then true else v`, on peut écrire `b || v`.

```
let rec pour_tous prop l =
  match l with
  | [] -> true
  | a :: l' -> prop a && pour_tous prop l'
;;

let rec il_existe prop l =
  match l with
  | [] -> false
  | a :: l' -> prop a || il_existe prop l'
```

```
;;
```

Question 7 Désolé, la fonction `suffixes` n'avait rien à faire ici.

```
let longueur l = List.fold_right (fun a len -> len + 1) l 0

let pour_tous p l =
  List.fold_right (fun x v -> p x && v) l true
;;

let il_existe p l =
  List.fold_right (fun x v -> p x || v) l false
;;
```

Notons que pour `pour_tous` et `il_existe`, ces versions sont moins efficaces que les précédentes, puisque l'on parcourt la liste jusqu'au bout dans tous les cas.

III. Combinaison à droite, combinaison à gauche

Question 8 Pour la fonction `miroir`, on appelle `myst1` avec la liste vide en premier argument :

```
let miroir l = myst1 [] l ;;
```

soit :

```
let miroir l = List.fold_left (fun m a -> a :: m) [] l ;;
```

ou encore :

```
let miroir l =
  let rec aux l m =
    match l with
    | [] -> m
    | a :: l' -> aux l' (a :: m)
  in
  aux l []
;;
```

La fonction de concaténation n'est autre que `myst2`. Cependant, reposant sur l'utilisation de `List.fold_right`, elle n'est pas récursive terminale.

On peut en avoir une version récursive terminale en utilisant `List.fold_left` ou, de façon similaire, en utilisant un renversement de liste (comme l'effet de `myst1`).

```
let concat l1 l2 = myst1 l2 (myst1 [] l1) ;;
```

IV. Listes représentant des ensembles

Question 9 C'est une variante de `il_existe`.

```
let rec appartient x l =
  match l with
  | [] -> false
  | a :: l' -> x = a || appartient x l'
;;
```

Un petit commentaire suite à la lecture de vos productions. Il y a une version avec un filtre `| [y] -> if x = y then true else false`. Notons tout d'abord que le test peut être avantageusement remplacé par le simple `x = y`. Sinon, ce filtre est-il vraiment utile ?

Question 10

```
let rec intersection l1 l2 =
  match l1 with
  | [] -> []
  | a :: l1' ->
      if appartient a l2 then
        a :: intersection l1' l2
      else
        intersection l1' l2
;;

let rec union l1 l2 =
  match l1 with
  | [] -> l2
  | a :: l1' ->
      if appartient a l2 then
        union l1' l2
      else
        a :: union l1' l2
;;
```

Question 11 Ces fonctions de complexité de l'ordre du produit des longueurs des deux listes, puisque l'on parcourt la seconde pour chaque élément de la première.

Question 12

```
let rec appartient x l =
  match l with
  | [] -> false
  | a :: l' -> if a < x then appartient x l' else a = x
```

```
;;

let rec intersection l1 l2 =
  match (l1, l2) with
  | [], _ -> []
  | _, [] -> []
  | a :: l1', b :: l2' ->
      if a < b then
        intersection l1' l2
      else if a = b then
        a :: intersection l1' l2'
      else
        intersection l1 l2'
;;

let rec union l1 l2 =
  match (l1, l2) with
  | [], _ -> l2
  | _, [] -> l1
  | a :: l1', b :: l2' ->
      if a < b then
        a :: union l1' l2
      else if a = b then
        a :: union l1' l2'
      else
        b :: union l1 l2'
;;
```

Cette fois-ci, on parcourt simultanément les deux listes et á chaque tour, on "consomme" un élément de l'une ou l'autre liste, en temps constant.

V. Quelques exercices supplémentaires

Question 13 Si la liste est de longueur n et qu'elle contient t fois `true` (ces deux informations pouvant être obtenus par un premier parcours), elle contient donc $n-t$ fois `false`.

Ainsi, si à un moment donné, on a lu k éléments de la liste et trouvé b `true`, on a passé $k-b$ `false` et il reste donc $(n-t)-(k-b)$ `false` dans ce qui reste à lire. On cherche donc à maximiser la quantité

$$b+(n-t)-(k-b)=2b+n-t-k$$

ou plus simplement $2b-k$ puisque ce sont les quantités qui varient lors du parcours.

```
let coupe l =
  let rec aux l k c p m =
    (* k correspond à la position dans la liste,
       c à 2 b - k,
       p à la position de coupe candidate
       et m le max de 2 b - k correspondant
    *)
    match l with
    | [] -> p
    | a :: l' ->
        if a then
          let c' = c + 1 in
          if c' > m then (* nouveau maximum *)
            aux l' (k + 1) c' (k + 1) c'
          else
            aux l' (k + 1) c' p m
        else (* a = false *)
          aux l' (k + 1) (c - 1) p m
  in
  aux l 0 0 0 0
;;
```

Question 14

```
let rec suffixes l =
  match l with
  | [] -> [ [] ]
  | a :: l' -> l :: suffixes l'
;;
```

Question 15 On remarque qu'une sous-liste de `a :: l` est soit de la forme `a :: l'` ou de la forme `l'` avec `l'` une sous-liste de `l`.

```
let rec sous_listes l =
  match l with
  | [] -> [ [] ]
  | a :: l2 ->
      let sl = sous_listes l2 in
      List.fold_left
        (fun sl' l' ->
          let nouvelle_sous_liste = a :: l' in
          nouvelle_sous_liste :: sl')
        sl sl
;;
```