TD3 - Solution

Yaëlle Vinçont

1 Programmation sur les listes

Question 1 Ecrire `intersection : 'a list -> 'a list -> 'a list` qui prend deux listes triées croissantes et renvoie leur intersection, triée croissante.

Exemple `intersection [1;3;5] [3;4;5;8] -> [3;5]`

Solution Idée de l'algorithme calculer l'intersection de l1 et l2 :
- On parcourt les deux listes en même temps
- Si l'une des deux est vide, on a fini
- Sinon (hd1 : :tl1 et hd2 : :tl2) on compare les éléments en tête :
  - si hd1 = hd2, on a un élément commun : on l'ajoute à notre intersection et on avance sur les 2 listes
  - si hd1 < hd2, on avance sur la liste 1 et pas la liste 2
  - sinon on avance sur la liste 2 et pas la liste 1

  Par exemple :
  - `intersection [3;4;5] [3;8;10]` -> hd1 = hd2 = 3 => 3 est dans l'intersection, et on calcule `intersection [4;5] [8;10]`
  - `intersection [1;3;5] [3;8]` -> hd1 < hd2 donc hd1 ne peut pas être dans l'intersection : on calcule `intersection [3;5] [3;8]`

Solution 1 : non terminale récursive

```
let rec intersection l1 l2 =
  match l1, l2 with
  (* si une des (ou les deux) listes est vide *)
  | [], _ | _, [] -> []
  | hd1::tl1, hd2::tl2 ->
    (* on ajoute hd1 a l'intersection des deux restes de liste *)
    if hd1 = hd2 then hd1::(intersection tl1 tl2)
    (* on avance sur l1 et pas l2 *)
    else if hd1 < hd2 then intersection tl1 l2
    (* on avance sur l2 et pas l1 *)
    else intersection l1 tl2
```

Solution 2 : terminale récursive

```
let intersection l1 l2 =
  let rec inter acc l1 l2 =
    match l1, l2 with
    | hd1::tl1, hd2::tl2 ->
      (* on ajoute hd1 a l'intersection et on continue sur le reste
       * des deux listes *)
      if hd1 = hd2 then inter (hd1::acc) tl1 tl2
      (* on avance sur l1 et pas l2 *)
      else if hd1 < hd2 then inter acc tl1 l2
      (* on avance sur l2 et pas l1 *)
      else inter acc l1 tl2
    (* on a traite le cas ou les deux listes sont non vides,
     * il reste le cas ou l'une des deux est vide :
     * on peut utiliser _, _ plutot que detailler *)
    | _, _ -> acc
  in
  List.rev (inter [] l1 l2)
```

NB : comme on crée l'intersection au fur et à mesure, on ajoute les éléments à la liste dans l'ordre où on la parcourt. Cependant, l'ajout se faisant toujours en tête de liste cela veut dire qu'on récupère la liste à l'envers. Il faut donc utiliser `List.rev : 'a list -> 'a list` pour la remettre à l'endroit.

Par exemple, si on a `intersection [1;3;5] [3;4;5;8]`, on va ajouter en tête de l'accumulateur d'abord 3 puis 5. On aura donc à la fin comme accumulateur : `5::3::[] = [5; 3]`. Et `List.rev [5; 3] = [3; 5]`.

Question 2 Avec un itérateur, écrire `duplique : 'a list -> 'a list` qui duplique tous les éléments de la liste.

Exemple `duplique [1;2;3] -> [1;1;2;2;3;3]`

Solution Idée de l'algorithme : parcourir la liste et pour chaque élément l'ajouter 2 fois en tête d'un accumulateur.

Comme on ajoute toujours en tête et que l'on veut conserver l'ordre, on va vouloir parcourir de droite à gauche -> `List.fold_right`.

```
let duplique l = List.fold_right (fun x acc -> x::x::acc) l []
```

Question 3 Écrire `nombre_pair_d_elements : 'a list -> bool` qui vérifie si une liste a un nombre pair d'éléments.

NB : on renomme `nombre_pair_d_elements` en `npde` pour des raisons de place.

Exemple `npde [] -> true, npde ['a'] -> false, npde [1;4] -> true, npde [1;8;4] -> false`

Solution en comptant Idée de l'algorithme : on compte le nombre d'éléments de la liste et on regarde s'il est pair.

```
let npde l = ((List.length l) % 2 = 0)
```

Cette solution renvoie fait le calcul, et vérifie si le reste modulo 2 est nul. On peut aussi recalculer la longueur nous-même :

```
let npde l =
  let rec len l =
  | [] -> 0
  | _::tl -> 1 + (len tl)
  in
  ((len l 0) % 2) = 0
```

Solution avec un booléen Idée de l'algorithme : comme le résultat est alternativement vrai ou faux, on peut utiliser un booléen sur lequel on applique not à chaque nouvel élément. Initialisé à true (pour la liste vide), il vaudra faux pour le premier élément, true pour le deuxième, etc ...

```
let npde l =
  let rec aux acc l =
  | [] -> acc
  | _::tl -> aux (not acc) tl
  in
  aux true l
```

On peut aussi faire le calcul avec un itérateur :

```
let npde l = List.fold_left (fun acc _ -> not acc) true l
```

NB : on présente ici 4 façons de faire, mais il y en a plus. On pourrait par exemple calculer la longueur en terminal récursif ou avec un itérateur ... ou faire une version non terminale avec le booléen.

Question 4 Avec un itérateur, écrire `second_max : 'a list -> 'a list` qui renvoie le 2ème plus grand élément d'une liste. [ ] donne une erreur, [v] donne v.

Exemple `second_max [3;1;5;2;9;0] -> 5, second_max [1;2;3;4] -> 3`

Solution Idée de l'algorithme :
- Parcourir la liste en mémorisant à chaque instant les deux plus grands éléments max et snd_max
- Pour chaque élément x :
  - si x > max, le maximum est x et le second maximum est max
  - sinon si x > snd_max, le maximum est max et le second maximum est x
  - sinon le maximum est max et le second maximum est snd_max
- Initialement, max et snd_max valent la première valeur de la liste
- Si la liste est vide, exception

```
let second_max l =
  (* on doit faire un match pour récupérer le premier élément,
   * et renvoyer une erreur s'il n'y en a pas *)
  match l with
  (* si la liste est vide, exception *)
  | [] -> failwith "Liste vide"
  | hd::tl ->
    snd
      (* on parcourt la liste avec un accumulateur *)
      (List.fold_left
         (* pour chaque element *)
         (fun (max, snd_max) x ->
            (* si x > max ... *)
            if x > max then x, max
            (* sinon si x > snd_max ... *)
            else if x > snd_max then max, x
            (* sinon ... *)
            else max, snd_max)
         (* initialement, max et snd_max ... *)
         (hd, hd)
         (* comme on a deja traite le premier element,
          * on peut commencer a la suite *)
         tl)
```

Principe de la compression RLE : on remplace les suites du même élément par une paire (élement, nombre d'éléments à la suite)

Question 5 Écrire `compression : 'a list -> ('a * int) list` qui compresse une liste selon la méthode RLE.

Exemple `compression ['a';'a';'a';'b';'b';'a'] -> [('a', 3); ('b', 2); ('a', 1)]`

Solution 1 Idée de l'algorithme :
- Parcourir la liste en retenant l'élément x qu'on est en train de compresser et son nombre d'occurences nb_x
- Si la liste est vide, on ajoute le dernier élément et son nombre d'occurence a la liste compressee
- Sinon (hd : :tl)
  - si x = hd on incrémente le compteur du nombre d'occurences de x et on appelle récursivement
  - sinon on ajoute la paire (x, nb_x) à la liste compressée, et maintenant x = hd et nb_x = 1
- Initialisation : x est la tête de la liste (s'il y en a une)

NB : on va devoir utiliser `List.rev` car la liste sera à l'envers

```
let compression l =
  let rec compr acc x occ_x l =
    match l with
    (* si la liste est vide ... *)
    | [] -> (x, occ_x)::acc
    (* sinon, si x = hd ... *)
    | hd::tl when x=hd -> compr acc x (occ_x+1) tl
    (* sinon ... *)
    | hd::tl -> compr ((x, occ_x)::acc) hd 1 tl
  in
  (* on recupère la première valeur *)
  match l with
  | [] -> []
  | hd::tl -> List.rev (compr [] hd 1 tl)
```

Solution 2 Idée de l'algorithme :
- Parcourir la liste en regardant 2 éléments à chaque fois et en gardant le nombre d'occurences du premier des 2
- Si la liste est vide, on renvoie l'accumulateur
- Si la liste contient un élément (cas qu'il faut gérer séparement si on veut regarder les éléments 2 par 2), on ajoute l'élément et son nombre d'occurences à l'accumulateur
- Sinon (x : :y : :s)
  - si x = y, alors on augmente le nombre d'occurences de x
  - sinon on ajoute la paire (x, nb_x) à l'accumulateur et on remet le compteur a 1 pour y

  et on se rappelle sur (y : :s)

```
let compression l =
  let rec compr l nb_x acc =
    match l with
    (* si la liste est vide ... *)
    | [] -> acc
    (* si la liste contient 1 élément ... *)
    | [x] -> (x, nb_x)::acc
    (* sinon *)
    | x::y::s ->
      (* si x = y ... *)
      if x = y then
        compr acc (nb_x + 1) (y::s)
      (* sinon ... *)
      else
        compr ((x, nb_x)::acc) 1 (y::s)
  in
  List.rev (compr [] 1 l)
```

Question 6 Ecrire `ajoute_a_liste : ('a * int) -> 'a list -> 'a list` tq `ajoute_a_liste (c, n) l` ajoute n fois c en tête de l

Exemple `ajoute_a_liste ('a', 3) ['b'; 'b'; 'a'] -> ['a'; 'a'; 'a'; 'b'; 'b'; 'a']`

Solution Idée de l'algorithme : on fait une fonction récursive qui ajoute c en tête de la liste et se rappelle en décrémentant le compteur

```
let ajoute_a_liste (c, n) l =
  let rec add acc k =
    if k = 0 then acc
    else add (c::acc) (k-1)
  in
  add l n
```

Question 7 Avec `ajoute_a_liste`, écrire `decompression : ('a * int) list -> 'a list`.

Exemple

```
decompression [('a', 2); ('b', 1); ('c', 3); ('a', 3); ('b', 2)]
-> ['a'; 'a'; 'b'; 'c'; 'c'; 'c'; 'a'; 'a'; 'a'; 'b'; 'b']
```

Solution Idée de l'algorithme : on parcourt la liste et pour chaque élément (c, n) on utilise `ajoute_a_liste` pour ajouter c n fois en tête de l'accumulateur

```
let decompression l = List.fold_right (fun x acc -> ajoute_a_liste x acc) l []
```

NB : comme on passe directement x et acc en paramètre à `ajoute_a_liste`, on peut faire la version suivante :

```
let decompression l = List.fold_right ajoute_a_liste l []
```

2 Génération d'arbres binaires

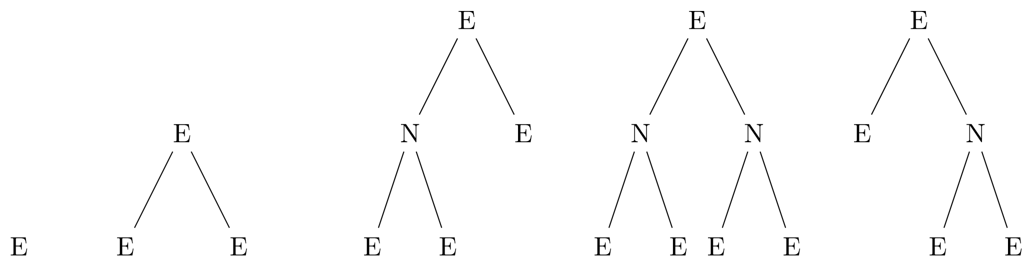
On génère tous les arbres binaires d'une hauteur donnée.

```
type t = E | N of t * t
```

Hauteur d'un arbre :

$$\begin{array}{lcl} \pi(E) & = & 0 \\ \pi(N(l,r)) & = & 1+\max(\pi(l),\pi(r)) \end{array}$$

Question 8 Dessiner tous les arbres dont la hauteur est strictement inférieure à 2 et ceux dont la hauteur est égale 2.



(a) Arbres de hauteur < 2 (b) Arbres de hauteur $= 2$

Le produit de deux listes d'arbres `produit : t list -> t list -> t list` est défini tel que :

```
produit [a1; ...; an] [b1; ...; bk] ->
[N(a1, b1);  ...; N(a1, bk); N(a2,  b1); ...; N(a2, bk); ...; N(an, b1); ...; N(an, bk)]
```

Question 9 Donner le résultat pour `produit [E; N(E, E)] [E; N(N(E, E), E)]`.

Solution `[N(E, E); N(E, N(N(E, E), E)); N(N(E, E), E); N(N(E, E), N(N(E, E), E))]`

Question 10 Ecrire `produit`. NB : l'ordre n'est pas important.

Solution Idée de l'algorithme pour le produit de trees1 et trees2 :
- Parcourir trees1
- Pour chaque élément t1 :
  - Parcourir trees2
  - Pour chaque élément t2 :
    - Créer N(t1, t2) et l'ajouter à l'accumulateur

```
let produit trees1 trees2 =
  (* parcourir trees1 *)
  List.fold_left
    (* pour chaque élément t1 *)
    (fun acc1 t1 ->
       (* parcourir trees2 *)
       List.fold_left
         (* pour chaque élément t2 *)
         (fun acc2 t2 ->
            (* créer N(t1, t2) ... *)
            N(t1, t2)::acc)
         (* on recupere ce que le premier parcours a deja accumule
          * comme valeur initiale de l'accumulateur *)
         acc1
         trees2)
    []
    trees1
```

NB : plutôt que de passer `acc1` comme accumulateur initial de la fonction du deuxième `fold_left`, on peut faire le `fold_left` avec un accumulateur initialisé à [] puis concaténer avec acc1 :

```
fun acc1 t1 ->
  let parcours2 =
    List.fold_left (fun acc2 t2 -> N(t1, t2)::acc) [] trees2
  in
  parcours2@acc1
```

Pour générer tous les arbres d'une hauteur donnée, on va utiliser deux fonctions mutuellement récursives `height:int -> t list` et `below: int -> t list` telles que `below h` renvoie la liste de tous les arbres de hauteur *strictement plus petite* que h et `height h` renvoie la liste de tous les arbres dont la hauteur est *exactement* h.

Question 11 Donner les définitions mutuellement récursives de `below` et `height`.

below pour calculer tous les éléments de profondeur strictement inférieures à une profondeur h, on peut calculer l'ensemble des éléments de profondeur n-1 ainsi que l'ensemble des éléments de profondeurs strictement inférieure à n-1, et concaténer ces deux ensembles.

height pour obtenir un arbre de profondeur n, il y a trois possibilités :
- avoir un arbre dont le sous-arbre gauche est exactement de profondeur n-1, et le sous-arbre droit de profondeur strictement inférieure à n-1
- avoir un arbre dont le sous-arbre droit est exactement de profondeur n-1 et le sous-arbre gauche de profondeur strictement inférieure à n-1
- avoir un arbre dont les deux sous-arbres sont exactement de profondeur n-1

Etant donnés deux ensembles de sous-arbres (par exemple ceux de profondeur n-1 et ceux de profondeur strictement inférieure à n-1) on utilise le produit pour créer l'ensemble des arbres dont le sous-arbre gauche vient du premier ensemble et le sous-arbre droit du deuxième

Notations $\otimes$ représente le produit comme défini dans la fonction ci-dessus tandis que @ représente la concaténation de deux listes.

$$\begin{aligned}
\text{below}(0) &= [\,] \\
\text{below}(n) &= \text{height}(n-1) \;@\; \text{below}(n-1) \\
\\
\text{height}(0) &= [\,E\,] \\
\text{height}(h) &= (\text{below}(n-1) \otimes \text{height}(n-1)) \\
&\quad @\; (\text{height}(n-1) \otimes \text{below}(n-1)) \\
&\quad @\; (\text{height}(n-1) \otimes \text{height}(n-1))
\end{aligned}$$

Question 12 Ecrire le code OCaml pour `below` et `height`.

NB On implémente directement le code ci-dessus, en utilisant la construction `let rec f x = ... and g x = ...` pour définir deux fonctions f et g mutuellement récursives.

```
let rec below h =
  (* below(0) *)
  if h = 0 then []
  (* below(h) *)
  else (height (h-1))@(below (h-1))

and height h =
  (* height(0) *)
  if h = 0 then [E]
  (* height(h) *)
  else
    let h_h1 = height (h-1) in
    let b_h1 = below (h-1) in
    (produit h_h1 b_h1) @ (produit b_h1 h_h1) @ (produit h_h1 h_h1)
```

NB On peut remarquer que cette façon de faire n'est pas du tout optimisée. En effet, si on veut calculer `height 2`, on peut représenter par l'arbre ci-dessous différents appels de fonction (a -> b signifie que le calcul de a fait appel à b)

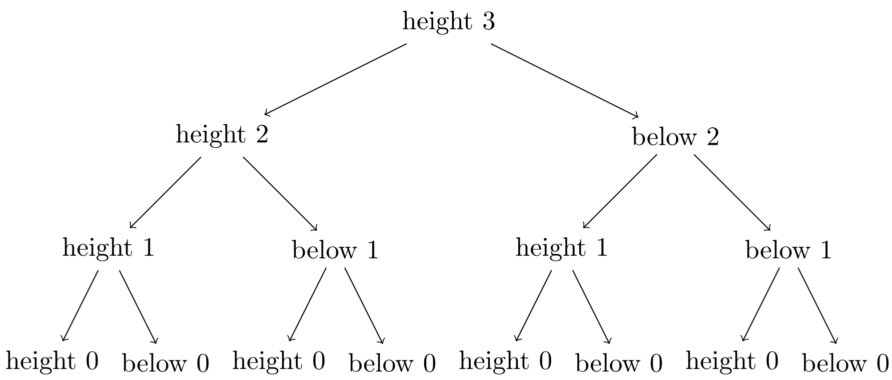


FIGURE 2 – Appels pour height 2

Déjà sur un simple appel avec une profondeur de 3, on peut remarquer que l'on va re-calculer quatre fois height 0 et below 0. Je vous laisse imaginer sur des profondeurs demandés plus grande ...

On verra par la suite qu'il est possible de retenir les calculs faits précédemment, technique qui s'appelle la mémoïzation, et qui permettrait ici de ne calculer qu'une fois chaque résultat et d'ensuite réutiliser le résultat déjà calculé.