OPTION INFORMATIQUE
Devoir surveillé 1 (2h)
Corrigé

Toutes les fonctions sont à écrire en OCaml. On attachera le plus grand soin à leur lisibilité. On pourra toujours librement utiliser une fonction demandée à une question précédente, même si cette question n'a pas été traitée. Lorsqu'une question impose une contrainte sur le coût en temps d'une fonction, on ne demande pas de démontrer que cette contrainte est respectée.

Pour toute fonction programmée, on précisera le type de cette fonction.

1 Représentation d'ensembles par des listes

Dans cette partie, on représente un ensemble fini par la liste **triée** (dans l'ordre croissant) **sans doublon** de ses éléments. On identifiera alors l'ensemble et la liste le représentant.

1. Écrire une fonction calculant le cardinal d'un ensemble.
   Corrigé :

```
let rec cardinal l =
    match l with
      [] -> 0
    | a::q -> 1 + cardinal q;;
```

2. Écrire une fonction renvoyant l'élément minimal d'un ensemble non vide.
   Corrigé :

```
let elem_min l = List.hd l;;
```

3. Écrire une fonction renvoyant l'élément maximal d'un ensemble non vide.
   Corrigé :

```
let rec elem_max l =
    match l with
      [] -> failwith "non défini"
    | [a] -> a
    | a::q -> elem_max q;;
```

4. Écrire une fonction testant l'appartenance d'un élément à un ensemble.
   Corrigé :

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

5. Écrire une fonction `comprehension` prenant en argument un prédicat p et un ensemble l et renvoyant l'ensemble des éléments de l vérifiant p.
   Corrigé :

```
let rec comprehension p l =
    match l with
      [] -> []
    | a::q -> if p a then a::comprehension p q else comprehension p q;;
```

6. Écrire une fonction `insertion` prenant en argument un élément x et un ensemble l et renvoyant l'ensemble contenant x et les éléments de l.
Corrigé :

```
let rec insertion x l =
    match l with
      [] -> [x]
    | a::q -> if a<x then a::(insertion x q)
                else if a=x then l
                else x::l;;
```

7. Écrire une fonction renvoyant l'union de deux ensembles. Chaque liste ne devra être parcourue qu'au plus une fois.
Corrigé :

```
let rec union l1 l2 =
    match l1,l2 with
      [],_ -> l2
    | _,[] -> l1
    | a1::q1,a2::q2 -> if a1 < a2 then a1::(union q1 l2)
                            else if a1 = a2 then a1::(union q1 q2)
                            else a2::(union l1 q2);;
```

8. Écrire sur le même principe une fonction renvoyant l'intersection de deux ensembles, une fonction renvoyant la différence ensembliste de deux ensembles, et une fonction testant l'inclusion d'un ensemble dans un autre.

```
let rec intersection l1 l2 =
    match l1,l2 with
      [],_ -> []
    | _,[] -> []
    | a1::q1,a2::q2 -> if a1 < a2 then intersection q1 l2
                            else if a1 = a2 then a1::(intersection q1 q2)
                            else intersection l1 q2;;


let rec difference l1 l2 =
    match l1,l2 with
      [],_ -> []
    | _,[] -> l1
    | a1::q1,a2::q2 -> if a1 < a2 then a1::(difference q1 l2)
                            else if a1 = a2 then difference q1 q2
                            else difference l1 q2;;

let rec inclus l1 l2 =
    match l1,l2 with
      [],_ -> true
    | _,[] -> false
    | a1::q1,a2::q2 -> if a1 < a2 then false
                            else if a1 = a2 then inclus q1 q2
                            else inclus l1 q2;;
```

9. On cherche à écrire une fonction `images` renvoyant l'ensemble des images des éléments d'un ensemble par une fonction f. L'enjeu est de s'assurer que ces images vont bien apparaître dans l'ordre croissant et sans doublon dans le résultat.

   (a) Écrire une version de cette fonction `images` utilisant la fonction `insertion`.
   Corrigé :

```
let rec images f l =
    match l with
      [] -> []
    | a::q -> insertion (f a) (images f q);;
```

(b) Donner un exemple de fonction f pour laquelle le calcul de la version précédente de `images` a un coût quadratique en la longueur de la liste.

Corrigé :

Il suffit de prendre une fonction f strictement décroissante, par exemple `f = fun n -> -n` pour un ensemble d'entiers. Chaque insertion doit alors parcourir l'intégralité de la liste d'images déjà formée. Il y a alors n(n-1)/2 comparaisons effectuées, d'où un coût quadratique.

(c) Pour améliorer le coût en moyenne, on se propose d'utiliser un tri rapide. Écrire une fonction `repartition` prenant en argument une liste l (non nécessairement triée ou sans doublon) et un pivot p, et renvoyant la liste des éléments de l strictement inférieurs au pivot et la liste des éléments strictement supérieurs au pivot.

Corrigé :

```
let rec repartition l p =
    match l with
        [] -> [], []
        |x::q -> let l1, l2 = repartition q p in
                    if x < p
                    then x::l1, l2
                    else if x > p
                    then l1, x::l2
                    else l1, l2;;
```

(d) En déduire une fonction récursive `tri_rapide` prenant en argument une liste non triée et renvoyant l'ensemble de ses éléments (sous forme de liste triée sans doublon)

Corrigé :

```
let rec tri_rapide l =
    match l with
      [] | [_] -> l
    | a::q -> let l1,l2 = repartition q a in
              union (tri_rapide l1) (a::(tri_rapide l2));;
```

(e) Utiliser cette fonction `tri_rapide` pour obtenir une nouvelle version de la fonction `images`.

Corrigé :

```
let images2 f l = tri_rapide (List.map f l);;
```

10. Écrire une fonction `produit` réalisant le produit cartésien de deux ensembles. On notera que l'ordre que Caml utilise sur les couples est l'ordre lexicographique : `produit [1; 2; 3] [4; 5]` doit renvoyer `[(1, 4); (1, 5); (2, 4); (2, 5); (3, 4); (3, 5)]`.

Corrigé :

```
let rec produit l1 l2 =
    let rec aux x l2 l =
        (*ajoute à l le produit [x]*l2 *)
        match l2 with
          [] -> l
        | a::q -> (x,a)::(aux x q l)
    in
    match l1 with
      [] -> []
    | a::q -> aux a l2 (produit q l2);;
```

11. Écrire une fonction `parties` renvoyant l'ensemble des parties d'un ensemble. Là encore, on notera que Caml utilise l'ordre lexicographique sur les listes, ainsi `parties [1; 2; 3]` doit renvoyer `[[]; [1]; [1; 2]; [1; 2; 3]; [1; 3]; [2]; [2; 3]; [3]]`.
Corrigé :
On raisonne de la façon suivante :
- si l est vide, sa seule partie est l'ensemble vide.
- si l contient un élément a, on considère les parties de l ne contenant pas a et celles contenant a. Le deuxième ensemble s'obtient en ajoutant a à chaque partie du premier ensemble.

```
let rec parties l =
    match l with
      [] -> [[]]
    | a::q -> let pq = parties q in
                union pq (images (fun r -> insertion a r) pq);;
```

2 Parcours d'arbre

On définit en Ocaml, pour représenter des arbres avec à chaque noeud un nombre quelconque de fils, le type suivant :

```
type 'a arbre = V | N of 'a * 'a arbre list;;
```

1. Lister sans justification les noeuds dans le parcours postfixe de l'arbre
`a = N (9, [N (1, []); N (2, [N (4, [N (1, []); N (3, [])])])])`
Corrigé :
1 - 1 - 3 - 4 - 2 - 9
2. Écrire une fonction `hauteur` calculant la hauteur d'un tel arbre (on rappelle que par convention l'arbre vide est de hauteur -1).
Corrigé :

```
let rec hauteur a =
    match a with
        V-> -1
        |N(x,f)->1 + hauteur_fratrie f
and hauteur_fratrie f =
    match f with
        []-> -1
        |a::q->max(hauteur a) (hauteur_fratrie q);;
```

3. Écrire une fonction `parcours_largeur` prenant en argument une procédure p et un arbre a, et appliquant p aux noeuds de a dans l'ordre d'un parcours en largeur (chaque niveau étant parcouru de gauche à droite).
Corrigé :

```
let rec parcours_niveau p a n =
    match a with
        V -> ()
        |N(x, l) ->
            if n = 0 then p x
            else parcours_niveau_fratrie p l (n-1)
and parcours_niveau_fratrie p l n =
    match l with
        [] -> ()
        |a::q -> parcours_niveau p a n;  parcours_niveau_fratrie p q n;;
```

```
let parcours_largeur p a =
    let h = hauteur a in
    for i = 0 to h do
        parcours_niveau p a i
    done;;
```

3 Minimum exclus

L'enjeu de cet exercice est de calculer le minimum exclus (ou **mex**) d'un tableau d'entiers naturels, c'est-à-dire le plus petit entier naturel qui n'apparaît pas dans le tableau. Par exemple, le mex de `[|2; 1; 0; 5|]` est 3, le mex de `[|4;7|]` est 0.

1. Donner sans justification le mex de `[|3; 6; 0|]`
   Corrigé :
   1

2. (a) Écrire une fonction `appartient` prenant en argument un tableau d'entiers t et un entier x, et testant si x est dans t.
   Corrigé :

   ```
   let appartient x t =
       let l = Array.length t in
       let i = ref 0 in
       while !i < l && t.(!i) <> x do
           incr i
       done;
       !i <> l;;
   ```

   (b) En déduire une fonction `mex1` calculant le mex d'un tableau d'entiers naturels.
   Corrigé :

   ```
   let mex1 t =
       let k = ref 0 in
       while appartient !k t do incr k done;
       !k;;
   ```

   (c) Déterminer la complexité, dans le pire cas et le meilleur cas, de la fonction précédente, en fonction de la longueur n du tableau et de son maximum m.
   Corrigé :
   `appartient` est en $O(n)$. Chaque passage dans le while est donc en $O(n)$.
   Dans le meilleur cas (quand 0 n'est pas dans t), on sort du while la première fois que la condition est évaluée, d'où une complexité en $O(n)$.
   Dans le pire cas (atteint quand t contient tous les éléments de 0 à son maximum), on passe $n(= m + 1)$ fois dans le while, d'où une complexité en $O(n^2)$ (pas besoin de l'exprimer en fonction de m, le sujet est trompeur).

3. (a) Écrire une fonction `presences` prenant un tableau t d'entiers naturels et renvoyant un tableau p de $m + 1$ booléens, où m est le maximum de t, tel que `p.(k)` indique si k est présent dans t.
   Corrigé :

   ```
   let maximum t =
       let m = ref t.(0) in
       for i = 1 to Array.length t - 1 do m := max !m t.(i) done;
       !m;;

   let presences t =
       let m = maximum t in
   ```

```
    let est_present = Array.make (m+1) false in
    for i = 0 to Array.length t - 1 do est_present.(t.(i)) <- true done;
    est_present;;
```

(b) En déduire une autre implémentation `mex2` du minimum exclus.

Corrigé :

```
let mex2 t =
    if Array.length t = 0 then 0 else begin
    let est_present = presences t in
    let m = Array.length est_present in
    let k = ref 0 in
    while !k <= m && est_present.(!k) do incr k done;
    !k
    end;;
```

(c) Déterminer la complexité dans le pire et le meilleur cas de `mex2`, toujours en fonction de n et m.

Corrigé :

`maximum` est en $O(n)$, `presences` est en $O(n+m)$ (à cause du `Array.make (m+1) false`, initialiser un tableau a un coût linéaire), donc `mex2` est en $O(n+m)$, dans le pire comme dans le meilleur cas.

(d) Démontrer formellement, à l'aide d'un invariant de boucle, la correction de `mex2` (en supposant que `presences` est correcte).

Corrigé :

On note A l'ensemble des entiers naturels qui ne sont pas dans t, et on pose

$$I : \forall x \in A, k \leq x$$

On montre que I est un invariant du `while` de `mex2` :

- Initialement, $k = 0$, donc I est vérifié.
- Supposons que I soit vrai au début d'une itération.
  A la fin de l'itération, en notant k' la nouvelle valeur de la variable, on a $k' = k + 1$.
  Puisque l'itération a lieu, la condition est vérifiée, donc k est présent dans t, donc $k \notin A$. On en déduit (d'après I) $\forall x \in A, k < x$, d'où $\forall x \in A, k + 1 \leq x$.
  I est donc toujours vrai en fin d'itération.
- En conclusion, I est vrai en sortie de boucle, mais on a alors $k > m$ ou `est_present(k)` faux. Dans les deux cas, $k \in A$, et d'après I, k est un minorant de A, c'est donc bien le minimum de A.

4. Écrire une variante `mex3` combinant la meilleure complexité dans le meilleur cas et la meilleure complexité dans le pire cas des versions précédentes, en utilisant un dictionnaire. On utilisera les appels suivants (tous de complexité constante) :
   - `hashtbl.create 0` renvoie un dictionnaire vide ;
   - `hashtbl.add h c v` ajoute au dictionnaire `h` la clé `c` associée à la valeur `v` ;
   - `hashtbl.mem h c` teste si la clé `c` apparait dans le dictionnaire `h`.

   On justifiera qu'on obtient bien les complexités désirées.

   Corrigé :

   On n'a en fait pas besoin des valeurs (toujours égales à 0 dans la suite), on utilise ici la structure de dictionnaire comme une façon efficace de manipuler l'ensemble formé des clés.

```
let mex3 t =
    let h = Hashtbl.create 0 in
    for i = 0 to Array.length t - 1 do
        Hashtbl.add h t.(i) 0;
    done;
    let k = ref 0 in
    while Hashtbl.mem h !k do incr k done;
    !k;;
```

La boucle `for` est en $O(n)$.

Chaque passage dans le `while` est en $O(1)$. On fait au moins un passage dans le while (quand 0 n'est pas dans t) et au plus n (quand t contient les éléments de 0 à son maximum).

Dans les deux cas, la complexité totale est en $O(n)$ (on a donc en fait gagné sur la complexité dans le pire cas par rapport aux deux versions précédentes, là encore le sujet est trompeur).