Ce sujet est composé de deux énoncés sur 8 pages : le premier (une page) concerne les grands entiers ; le second (7 page) parle d'arbres binaires de recherche et de B-arbres.

Votre code sera lu au maximum trois fois. S'il reste nébuleux après trois lectures, tant pis ! Il fallait expliquer et commenter votre code. Enfin, lorsqu'il est demandé une preuve, un baratin ne suffit pas...

Problème 1 - Grands entiers.

L'usage de références, de boucles n'est pas autorisé.

Les nombres que nous allons manipuler nécessitent une précision qui dépasse celle des entiers de la machine (type int). Nous allons donc commencer par définir une arithmétique de précision arbitraire. On se donne pour cela une base de calcul, par exemple base= 10000. La valeur de base importe peu et on supposera seulement qu'elle est paire, supérieure ou égale à 2 et que son double n'excède pas le plus grand entier machine. Un entier naturel de précision arbitraire est alors représenté par la liste de ses chiffres en base base, les chiffres les moins significatifs étant en tête de liste. Ainsi la liste $[1;2;3]$ représente l'entier $1 + 2 \times \texttt{base} + 3 \times \texttt{base}^2$. On définit le type nat suivant pour de tels entiers :

```
let base = ... ;;
type nat == int list ;;
```

Important ! Dans la suite, on garantira l'invariant suivant sur le type nat :
- tout élément de la liste est compris entre 0 et base-1, au sens large ;
- le dernier élément de la liste, lorsqu'il existe, n'est pas nul.

On notera que l'entier 0 est représenté par la liste vide.

Question 1) Définir une fonction cons_nat qui prend en argument un chiffre c (un entier machine, $0 \leqslant c < \texttt{base}$) et un grand entier n et qui renvoie le grand entier $c + \texttt{base} \times n$. La fonction cons_nat peut aider à garantir l'invariant du type nat

```
cons_nat : int -> nat -> nat
```

Question 2) définir une fonction add_nat qui calcule la somme de deux grands entiers. Indication : on pourra commencer par écrire une fonction prenant également une retenue en argument et appliquer l'algorithme traditionnel enseigné à l'école primaire.

```
add_nat : nat -> nat -> nat
```

Question 3) Définir une fonction cmp_nat qui prend en argument deux grands entiers n_1 et n_2 et qui renvoie un entier machine valant, -1 si $n_1 < n_2$, 1 si $n_1 > n_2$ et 0 si $n_1 = n_2$.

```
cmp_nat : nat -> nat -> int
```

Question 4) Définir une fonction sous_nat qui prend en argument deux grands entiers n_1 et n_2 et qui calcule la différence $n_1 - n_2$ en supposant $n_1 \geqslant n_2$. Indication : comme pour l'addition, on pourra commencer par écrire une fonction prenant également une retenue en argument.

```
sous_nat : nat -> nat -> nat
```

Question 5) Définir une fonction div2_nat qui prend en argument un grand entier n et qui calcule le quotient et le reste dans la division euclidienne de n par 2. Le quotient est un grand entier et le reste un entier machine valant 0 ou 1. On rappelle que la constante base est paire.

```
div2_nat : nat -> nat * int
```

Problème 2 - CCP 2012

Question 1)

Question 2)

Question 3)

Question 4)

Question 5)

Question 6)

Question 7)

Question 8)

Question 9)

Question 10)

Question 11)

Question 12)

Question 13)

Question 14)

Question 15)

Question 16)

Problème 1

Question 1)

```
let cons_nat c n =
  if c = 0 && n = [] then [] else c :: n;;
```

Question 2)

```
let rec add_aux n1 n2 r =
  match n1, n2 with
  | _,[] -> if r = 0 then n1 else add_aux [1] n1 0
  | [],_ -> add_aux n2 n1 0
  | c1 :: q1, c2 :: q2 ->
     let c' = (c1 + c2 + r) mod base and r' = (c1 + c2 + r) / base in
     cons_nat c' (add_aux q1 q2 r');;

let add_nat n1 n2 =
  add_aux n1 n2 0;;
```

Remarque : la retenue r vaut 0 ou 1.

Les deux cas de base sont évidents : l'un des deux entiers est nul et la retenue vaut 0.

Si $n_1 \neq 0$, $n_2 = 0$ et $r = 1$, alors on doit calculer $n_1 + 1$, qu'on peut voir comme la somme du grand entier n_1, du grand entier [1] et de la retenue 0.

Si $n_1 = 0$ et $n_2 \neq 0$, on utilise l'astuce classique qui consiste à relancer le calcul en intervertissant n_1 et n_2.

Dans le cas général, on écrit $n_1 = c_1 + \texttt{base}\, q_1$ et $n_2 = c_2 + \texttt{base}\, q_2$, où c_1, c_2 sont deux chiffres. Alors $n_1 + n_2 + r = (c_1 + c_2 + r) + \texttt{base}\,(q_1 + q_2)$. Mais $c_1 + c_2 + r$ peut dépasser la valeur de `base` : comme $c_1 + c_2 + r < 2\texttt{base}$, alors la division euclidienne de $c_1 + c_2 + r$ par `base` donne un quotient égal à 0 ou 1 (c'est la retenue r') et un reste c' tel que $0 \leqslant c' < \texttt{base}$ (qui est donc un chiffre), donc avec $c_1 + c_2 + r = c' + \texttt{base}\, r'$, on obtient $n_1 + n_2 + r = c' + \texttt{base}\,(q_1 + q_2 + r')$.

On a ainsi une définition récursive de l'addition de deux grands entiers et d'une retenue.

Question 3)

```
let rec cmp_nat n1 n2 =
  match n1, n2 with
  | [], [] -> 0
  | _, [] -> 1
  | [], _ -> -1
  | c1 ::: q1, c2 :: q2 ->
    let comp = cmp_nat q1 q2 in
    if comp <> 0 then comp
    else if c1 < c2 then -1
    else if c1 = c2 then 0
    else 1;;
```

Si l'un des deux entiers n_1 ou n_2 est nul, alors la comparaison est immédiate.

Dans le cas général, on écrit $n_1 = c_1 + \texttt{base}\, q_1$ et $n_2 = c_2 + \texttt{base}\, q_2$, où c_1, c_2 sont deux chiffres. Si $q_1 = q_2$, alors la comparaison se fait sur c_1 et c_2. En revanche, si $q_1 > q_2$ par exemple, alors $q_1 \geqslant q_2 + 1$, donc $\texttt{base}\, q_1 \geqslant \texttt{base}\, q_2 + \texttt{base} > \texttt{base}\, q_2 + c_2$, donc a fortiori, $n_1 = \texttt{base}\, q_1 + c_1 \geqslant \texttt{base}\, q_1 \geqslant \texttt{base}\, q_2 + \texttt{base} > n_2$: le résultat de la comparaison de $q_1$et q_2 suffit à déterminer celui de n_1 et n_2.

Question 4)

```
let rec sous_aux n1 n2 r =
  match n1, n2 with
  | _, [] -> if r = 0 then n1 else sous_aux n1 [1] 0
  | c1 :: q1, c2 :: q2 ->
     let c', r' =
       if c1 >= (c2 + r) then (c1 - c2 - r, 0)
       else (base + c1 - c2 - r, 1)
     in
     cons_nat c' (sous_aux q1 q2 r');;

let sous_nat n1 n2 =
  sous_aux n1 n2 0;;
```

Dans notre fonction sous_aux, on fait l'hypothèse que $n_1 \geqslant n_2 + r$.

Si n_2 est nul et $r = 0$, alors c'est évident. Si $n_2 = 0$ et $r = 1$, alors on fait comme dans la fonction add_aux.

Sinon on écrit $n_1 = c_1 + \mathsf{base}\, q_1$ et $n_2 = c_2 + \mathsf{base}\, q_2$, où c_1, c_2 sont deux chiffres. Alors $n_1 - (n_2 + r) = (c_1 - c_2 - r) + \mathsf{base}\,(q_1 - q_2)$. Or $n_1 \geqslant n_2$, donc $q_1 \geqslant q_2$, donc

si $c_1 - c_2 - r$ est directement un chiffre, on a directement une définition par appel récursif ;

sinon on écrit $n_1 - (n_2 + r) = (\mathsf{base} + c_1 - c_2 - r) + \mathsf{base}\,(q_1 - q_2 - 1)$: dans ce cas, on a $c_1 < c_2 + r \leqslant \mathsf{base}$ donc $-(c_2 + r) \leqslant c_1 - (c_2 + r) < 0$, donc $0 \leqslant \mathsf{base} - (c_2 + r) \leqslant \mathsf{base} + c1 - (c_2 + r) < \mathsf{base}$, donc $\mathsf{base} + c_1 - c_2 - r$ est un chiffre ; de plus, on a nécessairement $q_1 \geqslant q_2 + 1$, sinon $q_1 \leqslant q_2$ donc $q_1 = q_2$ puis $n_1 = c_1 + \mathsf{base}\, q_1 = c_1 + \mathsf{base}\, q_2 = c_2 + \mathsf{base}\, q_2 + (c_1 - c_2) < c_2 + \mathsf{base}\, q_2 + r = n_2$: contradiction

donc on a encore un calcul récursif de la soustraction.

Question 5)

```
let rec div2_nat n =
  match n with
  | [] -> [],0
  | c :: m ->
      let q',r' = div2_nat m in
      let x = c + r' * base in
      let c', s' = (x / 2, x mod 2) in
      cons_nat c' q', s';;
```

Si $n = 0$, alors c'est évident.

Sinon $n = c + \mathsf{base}\, m$, puis on effectue la division euclidienne de m par 2 : $m = 2q' + r'$ avec $0 \leqslant r' \leqslant 1$, donc $n = c + \mathsf{base}\, r' + 2(\mathsf{base}\, q')$:

on effectue la division euclidienne de $c + \mathsf{base}\, r'$ par 2 : $c + \mathsf{base}\, r' = 2c' + s'$ avec $0 \leqslant s' \leqslant 1$, donc finalement

$n = 2(c' + \mathsf{base}\, q') + s'$: c'est la division euclidienne de n par 2.

Remarque : l'énoncé rappelait que la base est paire, franchement je ne vois pas à quoi ça sert.

Problème 2

Question 1)

```
let rec ajoutSequence v s =
  match s with
  | [] -> [v]
  | x :: q -> if x < v then x :: ajoutSequence v q else v :: s;;
```

Question 2) La complexité $C(n)$ de la fonction selon la longueur de la liste vérifie $C(0) = 1$ et $C(n) = 1 + C(n-1)$ (on ne compte que les appels de fonction), donc $C(n) = O(n)$.

Question 3) La fonction coupe s n calcule un couple de deux listes : la première est le préfixe de longueur n de la liste s, la seconde est sa complémentaire.

La fonction scissionSequence s calcule d'abord la longueur (fonction récursive intégrée) de la liste et appelle la fonction précédente avec les bons arguments ; enfin, sur la deuxième liste, elle sépare tête et queue.

```
let rec coupe s n =
  if n = 0 then [], s
  else match s with
       | [] -> failwith "n␣trop␣grand"
       | x :: q -> let s',t' = coupe q (n-1) in x :: s', t';;

let rec scissionSequence s =
  let n = (list_length s - 1) / 2 in
  let s1, t1 = coupe s n in
  let v= hd t1 and s2 = tl t1 in
  s1,v,s2;;
```

Question 4) Pour simplifier les notations, je note V pour Vide et N pour Noeud

```
eliminer 2 exemple
--> g = N(N(V,1,V), 2, V), vp = 2, d = N(V, 3, V)
    vp = v donc
    rg = eliminer 2 g
        --> g = N(V, 1, V), vp =2, d = V
            vp = v donc
            rg = eliminer 2 g
            --> g = V, vp = 1, d = V
                vp <= v donc
            <-- N(V, 1, eliminer 2 V)
                -->
                <-- V
            rg = N(V, 1, V)
            puis (vm, gp) = aux N(V, 1, V)
                -->
                <-- 1, V
        <-- N(V, 1, V)
     rg = N(1, V, 1)
     puis (vm, gp) = aux N(V, 1, V)
         -->
         <-- 1, V
    <-- N(V, 1, V)
<-- N(V, 1, N(V, 3, V))
```

Question 5) On montre d'abord que aux a calcule le couple (vr, ar) où m est la valeur maximale dans l'ABR a et ar est un ABR qui contient les mêmes étiquettes que a sauf ce maximum vr.

Par induction structurelle : soit a un ABR non vide,

- si a = N(g, m, V), alors aux a = (m, V) : comme a est un ABR, sa plus grande valeur est donc ici sa racine, donc la proposition est vraie pour le cas de base.
- sinon on calcule aux d : par hypothèse d'induction, on obtient le plus grand élément vr de d (donc de a) et un ABR ar ayant les mêmes étiquettes sauf ce maximum, puis on construit le couple vr, N(g,v,ar) ; le premier objet du couple est bien le maximum de a, le second est un ABR, car
  - g est un ABR
  - par hypothèse d'induction, ar est un ABR
  - les étiquettes de g sont toutes plus petites que v (puisque a est un ABR) et les étiquettes de ar sont les mêmes que celles de d sauf une, donc elles sont toutes plus grandes que v

donc la proposition est vraie pour a

D'après le principe d'induction structurelle, elle est vraie pour tout ABR.

Puis de même, on montre par induction structurelle que si a est un ABR, alors r = eliminer v a est un ABR dont les étiquettes sont celles de a sauf celles égales à v.

Par induction structurelle : soit a un ABR non vide,

- si a = V, alors r = V : la proposition est vraie pour le cas de base.
- sinon on écrit a sous la forme N(g, vp, d) ;
  - si v < vp, alors r = N(eliminer v g, vp, d),
    par hypothèse d'induction, comme g est un ABR, r' = eliminer v g est un ABR qui ne contient plus l'étiquette v et dont les étiquettes sont toutes les autres parmi celles de g : comme a est un ABR, ces étiquettes sont donc plus petites que vp, donc r est un ABR ;
    de plus, comme v < vp, les étiquettes de d sont toutes plus grandes que vp donc que v, donc aucun étiquette de r n'est égale à v et les étiquettes de r sont bien celles de a privées de v ;
  - si v > vp, de même en échangeant les rôles des sous-arbres droit et gauche
  - v = vp, alors les étiquettes de d sont toutes plus grandes strictement que v ; par hypothèse d'induction, rg est un ABR obtenu à partir de g en supprimant les étiquettes égales à v et seulement celles-là ; donc
    - si rg est vide, alors toutes les étiquettes de g sont égales à v et aucune étiquette de d n'est égale à v donc r = d est bien un ABR obtenu en supprimant dans a les étiquettes égales à v et seulement celles-là ;
    - sinon rg n'est pas vide, on calcule son plus grand élément vm et son ABR complémentaire dans rg (par appel à la fonction aux), puis on construit r = N(qp, vm, d) : toutes les étiquettes de gp sont plus petites que vm et sont toutes prises parmi celles de g donc elles sont toutes plus petites que vm, qui lui-même est plus petit que les étiquettes de d (puisque a est un ABR) et gp, d étant deux ABR, r est bien un ABR ;de plus, on a supprimé dans g uniquement les étiquettes égales à v et on a supprimé la racine, remplacée par vm, donc r est bien un ABR obtenu en supprimant dans a les étiquettes égales à v et seulement celles-là ;

  donc dans tous les cas, la proposition est vraie pour a

D'après le principe d'induction structurelle, elle est vraie pour tout ABR.

Question 6) En dehors des cas de base, les appels récursifs dans la fonction aux ont pour arguments des arbres de taille strictement inférieures dans $\mathbb{N}$ à celle de l'arbre initial, donc comme $\mathbb{N}$ est bien fondé, la fonction aux termine.

Et il est est de même pour la fonction eliminer, dont les appels récursifs vérifient la même propriété et qui fait appel à la fonction aux qui termine.

Question 7) Si a est de la forme N(V,v+1,d), alors la fonction eliminer ne fait qu'un seul appel récursif et calcule a lui-même. La complexité minimale est donc constante : $C_{min}(n) = O(1)$.

Si a est de la forme N(g,w,V) avec g un peigne gauche ne contenant pas v, alors la fonction eliminer est appelée n fois où n est la longueur de la branche g, donc la complexité est alors maximale : $C(n) = O(n)$.

Question 8) L'appel récursif de la fonction aux se fait toujours sur le fils droit, donc le nombre d'appels récursifs est la longueur de la branche droite de l'arbre, qui est majorée par la hauteur de l'arbre. La complexité de aux est donc linéaire selon la hauteur de l'arbre.

L'appel récursif de la fonction eliminer se fait toujours sur le fils gauche ou le fils droit mais pas les deux, donc le nombre d'appels récursifs est la longueur d'une branche menant de la racine à une feuille, qui est majorée par la hauteur de l'arbre.

Question 9)

```
let estComplet a =
  match a with
  | Feuille(l) -> list_length l + 1 = 2 * ordre
  | Noeud(b,f) -> list_length f + 1 = 2 * ordre;;
```

Question 10)

```
let etiquettes f = map fst f;;
let arbres f = map snd f;;

let rec somme l =
  match l with
  | [] -> 0
  | x :: q -> x + somme q;;

let rec taille a =
  match a with
  | Feuille(l) -> list_length l
  | Noeud(b,f) -> taille b + list_length f + somme (map taille (arbres f));;
```

Question 11)

```
let rec rechercheFrere v f =
  match f with
  | [] -> failwith "liste des freres vide !"
  | [e,a] -> a
  | (e,a) :: (e',a') :: q ->
      if v < e' then a else rechercheFrere v ((e',a') :: q);;

let rec recherche v a =
  match a with
  | Feuille(l) -> mem v l
  | Noeud(a1,f) -> mem v (etiquettes f)
                   || ( let (e1,a2) = hd f in
                        if v < e1 then recherche v a1
                        else
                        let bon_arbre = rechercheFrere v f in
                        recherche v bon_arbre) ;;
```

Question 12)

```
let rec scissionBArbre a =
  match a with
  | Feuille s -> let s1,e,s2 = scissionSequence s in Feuille(s1),e,Feuille(s2)
  | Noeud (b,f) -> let f1, p, f2 = scissionSequence f in
      let a1 = Noeud (b, f1) and (e, c) = p in
      let a2 = Noeud (c, f2) in
      (a1, e, a2) ;;
```

Question 13) Le parcours commence à la racine qui est noeud complet et qui est donc scindé en l'arbre Noeud(a1,[(12 ;b1)]) avec a1=Noeud(a2,[(9 ;b2)]).

Comme $3 < 12$ le parcours se dirige sur la branche de gauche c'est à dire l'arbre a1 dont la racine est incomplète (une seule étiquette 9) donc pas de scission.

Puis $3 < 9$ et le parcours se dirige sur l'arbre a2=Noeud(Feuille([2 ;4 ;6]),[(7,b3)]).

Comme $3 < 7$ le parcours atteint la feuille Feuille([2 ;4 ;6]) qui est complète et est donc scindée en Noeud(Feuille([2]),[(4,Feuille([6]))]).

Puis enfin $3 < 4$ et 3 est inséré dans la feuille de gauche qui devient Feuille([2 ;3]).

Question 14) L'algorithme est un parcours en profondeur de l'arbre : à chaque étape, on passe d'un arbre à un de ses fils, qui est de taille strictement inférieure dans $\mathbb{N}$. Comme $\mathbb{N}$ est bien fondé, l'algorithme termine.

Question 15) La preuve se fait encore par récurrence sur la taille de l'arbre (la fonction scissionBArbre étant correcte).

Question 16)

```
let rec remplacer x y l =
  match l with
  | [] -> []
  | (e,a) :: q -> if a = x then (e,y) :: q else (e,a) :: remplacer x y q;;

let rec ajouter v a =
  if estComplet a then
     let (a1, e , a2) = scissionBArbre a in
     let a' = Noeud(a1, [e,a2]) in
     ajouter v a'
  else
  match a with
  | Feuille(l) -> Feuille(ajoutSequence v l)
  | Noeud(a1, f) ->  let (e1,a2) = hd f in
                              if v < e1 then Noeud(ajouter v a1, f)
                              else
                              let ba = rechercheFrere v f in
                              let na = ajouter v ba in
                              let nf = remplacer ba na f in
                              Noeud(a1, nf);;
```