X 2009 - Informatique
Un corrigé.

1 Grands entiers.

Q.1. Si $n = \sum_{i=0}^{k} n_i b^i$ alors $c + bn = \sum_{i=0}^{k} m_i b^i$ avec $m_0 = c$ et $\forall i \geq 1,\ m_i = n_{i-1}$. L'opération proposée est donc juste une opération de consage. Il faut juste être attentif à ne pas ajouter un 0 à une liste vide pour garantir l'invariant du type `nat`

```
let cons_nat c n = if c<>0 or n<>[] then c::n else [];;
```

Q.2. Comme le propose l'énoncé, on écrit une première fonction prenant aussi en argument une retenue (qui vaut 0 ou 1).

```
add_nat_ret : nat -> nat -> int -> nat
```

Dans le cas "général", on additionne les unités et la retenue (on reste dans l'ensemble des entiers machines acceptables) ce qui donnera l'unité de la somme et une éventuelle retenue à propager. La fonction demandée s'en déduit avec un appel initial avec une retenue nulle.

```
let rec add_nat_ret a b r =
  match (a,b) with
  ([],[]) -> if r=0 then [] else [r]
 |(u::q,[]) -> if u+r < base then (u+r)::q
               else 0::(add_nat_ret q b 1)
 |([],u::q) -> if u+r < base then (u+r)::q
               else 0::(add_nat_ret a q 1)
 |(ua::qa,ub::qb) -> if (ua+ub+r)<base then (ua+ub+r)::(add_nat_ret qa qb 0)
               else (ua+ub+r-base)::(add_nat_ret qa qb 1) ;;

let add_nat a b = add_nat_ret a b 0 ;;
```

Q.3. Il faut comparer les bits de poids fort d'abord. Donc si $n_i = u_i + \texttt{base} \times q_i$, on comparer les entiers q_i et, en cas d'égalité, les entiers u_1.

```
let rec cmp_nat n1 n2 =
  match (n1,n2) with
  [],[] -> 0
 |_,[] -> 1
 |[],_ -> -1
 |u1::q1,u2::q2 -> let i=cmp_nat q1 q2 in
              if i=0 then begin
                    if u1<u2 then -1
                    else if u1=u2 then 0
                    else 1 end
              else i ;;
```

Q.4. La stratégie est la même que pour l'addition mais c'est cependant plus compliqué si on veut garantir que le dernier élément de la liste résultat est non nul. Par exemple, en base 10, quand on effectue $237 - 137$ ou $137 - 137$, le 0 des unités $(7 - 7)$ doit ou non apparaître. Ce problème est réglé par l'utilisation de la fonction de la question 1.

Ceci étant, la fonction auxiliaire `sous_nat_ret` prend en argument deux grands entiers n_1 et n_2 et une retenue r (qui vaut 0 ou 1) et renvoie le grand entier $n_1 - n_2 - r$.

```
let rec sous_nat_ret a b r =
  match (a,b) with
  ([],[]) -> if r=0 then [] else echouer "arguments"
 |(u::q,[]) -> if u-r >= 0 then cons_nat (u-r) q
               else cons_nat (u-r+base) (sous_nat_ret q b 1)
 |([],u::q) -> echouer "arguments"
```

```
 |(ua::qa,ub::qb) -> if (ua-ub-r)>=0 then
                              cons_nat (ua-ub-r)(sous_nat_ret qa qb 0)
                   else  cons_nat (ua-ub-r+base) (sous_nat_ret qa qb 1) ;;

let sous_nat a b = sous_nat_ret a b 0 ;;
```

Q.5. On adopte une stratégie récursive. Dans l'étape récurrente, on suppose que $n = n_0 + \texttt{base} \times m$. L'appel récurrent permet de connaître le grand entier q_m et le reste r_m tels que $m = 2q_m + r_m$. On a alors

$$n = n_0 + 2(\texttt{base} \times q_m) + (\texttt{base} \times r_m)$$

On distingue deux cas :

- Si $r_m = 0$ alors $n = (n_0 \ mod \ 2) + 2(\frac{n_0}{2} + \texttt{base} \times q_m)$ (la division est une division entière). Ceci est la division euclidienne cherchée et le quotient est $\frac{n_0}{2} + \texttt{base} \times q_m$ qui s'écrit comme grand entier en consant $\frac{n_0}{2}$ à q_m (comme en question 1).
- Si $r_m = 1$ alors $n = (n_0 \ mod \ 2) + 2(\frac{n_0 + \texttt{base}}{2} + \texttt{base} \times q_m)$ et on conclut pareillement.

On nutilise encore `cons_nat` pour garantir l'invariant sur le type `nat`.

```
let rec div2_nat n =
  match n with
  [] -> [],0
 |n0::m -> let (qm,rm)=div2_nat m in
           if rm=0 then (cons_nat (n0/2) qm) , (n0 mod 2)
           else (cons_nat ((n0+base)/2) qm) , (n0 mod 2) ;;
```

Q.6. Il suffit de renvoyer un objet identique au signe près.

```
let neg_z n = { signe = - n.signe ; nat = n.nat } ;;
```

Q.7. On veut calculer la somme $a+b$ connaissant $|a|, |b|$ et les signes de a et b. Selon que les signes sont égaux ou différents on doit additionner ou soustraire $|a|$ et $|b|$. Dans ce deuxième cas, le signe obtenu dépend de la position de $|a|$ par rapport à $|b|$.

```
let add_z a b =
  match (a.signe,b.signe) with
  (1,1) -> { signe = 1 ; nat = (add_nat a.nat b.nat)}
 |(-1,-1) -> { signe = -1 ;  nat = (add_nat a.nat b.nat)}
 |(1,-1) -> let s = cmp_nat a.nat b.nat in
            if s=0 then {signe = 1 ; nat = []}
            else if s=1 then {signe = 1 ; nat = (sous_nat a.nat b.nat)}
            else {signe = -1 ; nat = sous_nat b.nat a.nat}
  |  _   -> let s = cmp_nat a.nat b.nat in
            if s=0 then {signe = 1 ; nat = []}
            else if s=1 then {signe = -1 ; nat = (sous_nat a.nat b.nat)}
            else {signe = 1 ; nat = (sous_nat b.nat a.nat)}  ;;
```

Q.8. Le plus simple est de remarquer que $2^p u = (2^{p-1}u) + (2^{p-1}u)$ et d'adopter une stratégie récursive.

```
let rec mul_puiss2_z p u =
  if p=0 then u
  else let v=mul_puiss2_z (p-1) u in add_z v v ;;
```

Q.9. Là encore, on procède récursivement. Si $u = 2q+r$ avec $r = 1$ alors u est impair et on renvoie $(u, 0)$. Sinon $r = 0$ et un appel récursif donne v et p tels que $q = 2^p v$; on a alors $u = 2^{p+1}v$.

```
let rec decomp_puiss2_z u =
  let (q,r) = div2_nat u.nat in
  if r=1 then (u,0)
  else let (v,p) = decomp_puiss2_z {signe =u.signe;nat=q} in (v,p+1) ;;
```

2 Nombres dyadiques.

Q.10. Si $d = a2^b$ avec a impair ou nul alors $\frac{d}{2} = a2^{b-1}$ et a est toujours impair ou nul.

```
let div2_dya d = { m = d.m ; e = d.e - 1 } ;;
```

Q.11. On suppose que $d_1 = a_1 2^{b_1}$ et $d_2 = a_2 2^{b_2}$ avec a_i impair ou nul. Le cas où l'un des a_i est nul est simple ($d_1 + d_2$ vaut d_1 ou d_2). Sinon,

- si $b_1 > b_2$ alors $d_1 + d_2 = (2^{b_1 - b_2} a_1 + a_2)2^{b_2}$ est une forme convenable (la mantisse est bien impaire) ;
- si $b_1 < b_2$ la situation est symétrique ;
- sinon, $d_1 + d_2 = (a_1 + a_2)2^{b_1}$ et il suffit de décomposer $a_1 + a_2$ avec la question 9 pour obtenir la forme voulue. Ceci n'est possible que si $a_1 + a_2 \neq 0$ et on met donc un cas à part.

```
let add_dya d1 d2 =
     if d1.m.nat=[]  then d2
else if d2.m.nat=[] then d1
else if d1.e > d2.e then { m = add_z (mul_puiss2_z (d1.e-d2.e) d1.m) d2.m ;
                           e = d2.e }
else if d1.e < d2.e then { m = add_z (mul_puiss2_z (d2.e-d1.e) d2.m) d1.m ;
                           e = d1.e }
else let s = add_z d1.m d2.m in
     if s.nat=[] then { m = s ; e = 0}
     else let (u,p) = decomp_puiss2_z s in
          { m = u ; e = d1.e + p } ;;
```

Q.12. $d_1 - d_2 = d1 + (-d_2)$ doit faire l'affaire.

```
let sous_dya d1 d2 =
   add_dya d1 {m=neg_z d2.m ; e=d2.e} ;;
```

3 Listes à deux bouts.

Q.13. On peut, au choix, regarder si les listes droite et gauche sont vides ou si leurs longueurs sont nulle.

```
let ldb_est_vide l = (l.lg=0) & (l.ld=0) ;;
```

Q.14. Si g n'est pas vide, on en prend le premier élément. Sinon, comme la LDB est non vide et vérifie l'invariant (2) alors d possède un unique élément et c'est celui recherché.

```
let premier_g l = if l.lg<>0 then hd (l.g) else (hd l.d) ;;
```

Q.15. Il suffit de permuter g et d (l'invariant reste satisfait).

```
let inverse_ldb l = { lg = l.ld ; g = l.d ; ld = l.lg ; d = l.g } ;;
```

Q.16. J'écris une première fonction auxiliaire

`choisir : int -> 'a list -> 'a list -> 'a list`

prenant en argument un entier p et deux listes $l_1 = [a_1; \ldots; a_n]$ et $l_2 = [b_1; \ldots; b_m]$ telles que $n \geq p$. La fonction renvoie le couple de listes $([a_1; \ldots; a_p], [b_1; \ldots; b_m; a_n; \ldots; a_{p+1}])$. On utilise une stratégie récurrente simple. On a utilisé ici l'opérateur infixe de concaténation @ et la fonction `rev` (correspondant à `inverser` de l'énoncé).

```
let rec choisir p l1 l2 =
  if p=0 then ([],l2 @ (rev l1))
  else let (m1,m2)=choisir (p-1) (tl l1) l2 in
            ((hd l1)::m1,m2) ;;
```

On utilise alors l'indication de l'énoncé (on suppose c défini globalement) en renvoyant une LDB dont les tailles sont égales à une unité près. Selon que g ou d est la plus grande des liste, on reverse des éléments de l'une dans l'autre grâce à `choisir`.

```
let invariant_ldb l =
  if (l.lg <= c*l.ld + 1) & (l.ld <= c*l.lg + 1) then l
```

```
else let demi=(l.lg + l.ld)/2 in
     if l.lg>=l.ld then let (m1,m2)=choisir demi l.g l.d in
                        { lg=demi ; g=m1 ; ld=l.lg+l.ld-demi ; d=m2}
     else let (m1,m2)=choisir demi l.d l.g in
          { ld=demi ; d=m1 ; lg=l.lg+l.ld-demi ; g=m2} ;;
```

On notera que le coût d'une concaténation est proportionnel à la longueur de la liste de gauche concaténée et que le coût d'une image miroir est proportionnel à la longueur de la liste. On respecte donc la contrainte de complexité imposée par l'énoncé.

Q.17. Il suffit de conser un élément à la liste de gauche et d'assurer l'invariant avec la fonction de la question précédente.

```
let ajoute_g x l = invariant_ldb {lg=l.lg + 1; g = x::l.g; ld =l.ld; d=l.d };;
```

Q.18. On doit enlever l'élément en tête de la liste de gauche sauf si celle-ci est vide (mais dans ce cas, comme (2) a lieu, la LDB n'a qu'un élément et on doit renvoyer la LDB vide).

```
let enleve_g l = if l.lg=0 then {lg=0;g=[];ld=0;d=[]}
            else invariant_ldb {lg=l.lg - 1; g = (tl l.g); ld =l.ld; d=l.d };;
```

Q.19. La liste gauche grossit jusqu'à ce que (2) ne soit plus vérifié. Elle se deverse alors partiellement dans la liste droite (jusqu'à équilibrer les tailles) et le processus recommence.

Notons g_k et d_k les tailles des listes gauche et droite après le k-ième déversement. On a donc $g_1 = d_1 = 1$. De plus, le déversement suivant aura lieu quand la liste gauche dépassera la taille $3d_k + 1$ et on aura donc $g_{k+1} = d_{k+1} = 2d_k + 1$ (on dispose de $3d_k + 1 + d_k = 4d_k + 1$ éléments, on en ajoute un qui provoque un déversement qui fait un partage équitable). Une récurrence donne alors $d_k = g_k = 2^k - 1$. Il y a donc déversement quand on ajoute un élément de numéro $d_k + g_k = 2^{k+1} - 2$ (pour le second, le sixième, le quatorzième etc.) et ce déversement a un coût de l'ordre de 2^{k+1} opérations.

Supposons que $N = 2^p - 2$ (avec $p \geq 2$). Le nombre total d'opération est donc de l'ordre de

$$N + \sum_{k=1}^{p-1} 2^{k+1} = N + \frac{2^{p+1} - 4}{2 - 1} = O(N)$$

Le coût moyen d'un ajout est donc constant (dans le cas où N n'est plus de la forme $2^p - 2$, on peut l'encadrer par deux quantités de ce type...).

4 Polynômes de Bernstein.

Q.20. Une récurrence de construction immédiate montre que

$$\forall i, k \in \mathbb{Z},\ \forall x \in [0,1],\ B_i^k(x) \geq 0$$

On en déduit alors que $I(p)(x)$ est positif pour $x \in [0,1]$ si les coefficients de p sont positifs.

Un calcul simple donne

$$I(\langle 2; -1; 2\rangle) = 2B_0^2 - B_1^2 + 2B_2^2 = 2(1-X)^2 - 2X(1-X) + 2X^2 = 2 - 6X + 6X^2$$

et ce polynôme prend des valeurs positives sur $\mathbb{R}$ (son discriminant est strictement négatif). On n'a donc pas la réciproque à la propriété prouvée.

Q.21. Soit $p = \langle 2; -1; 2\rangle$. `raffine_g` p et `raffine_d` p étant à coefficients positifs, il prennent des valeurs positives sur $[0,1]$. D'après les résultats (3) et (4) p prend des valeurs positives sur $[0, 1/2]$ (avec (3)) et sur $[1/2, 1]$ (avec (4)). On retrouve ainsi le fait que p prend des valeurs positives sur $[0,1]$.

Q.22. D'après les résultats (3) et (4), pour tester la positivité sur $[0,1]$ d'un polynôme p, il suffit de tester celle de ses raffinements gauche et droit. Il s'agit bien d'une stratégie diviser pour régner puisque l'étude du raffinement gauche (ou droit) correspond à l'étude du polynôme initial sur

un demi-intervalle. On écrit une fonction auxiliaire de test prenant en argument un polynôme et un entier correspondant à une "profondeur de décomposition". Moralement, quand le second paramètre prend la valeur k, cela correspond à l'étude du polynôme initial sur un intervalle de longueur $1/2^k$ (et il y a bien 2^k tels appels puisque le nombre de test est multiplié par 2 à chaque étape). On suppose définie globalement une constante `limite` donnant la valeur limite de la profondeur de décomposition.
Pour le cas de base, on a besoin d'une autre fonction `coeff_pos` testant la positivité des coefficients d'un élément de type `dya list` (une condition suffisante pour que le polynôme soit positif sur $[0,1]$ est que les coefficients de la LDB le soient).

```
let rec coeff_pos l =
   match l with
   [] -> true
  |d::q -> ((d.m.signe = 1) or (d.m.nat=[]))  (* d est positif ou nul *)
         & (coeff_pos q) ;;   (* test sur les autres \'el\'ements *)

let test_pos p =
  let rec test_aux q k =
     if k=limite then (coeff_pos p.g)&(coeff_pos p.d)
     else (test_aux (raffine_g q) (k+1)) & (test_aux (raffine_d q) (k+1))
in test_aux p 0 ;;
```

Q.23. De manière immédiate, on a

$$\texttt{derive}(c.p) = c.\texttt{derive}\ p$$

On montre par récurrence sur la taille du polynôme p que

$$\forall p,\ \forall c, d \in \mathcal{D},\ \texttt{integre}\ d\ (c.p) = c.(\texttt{integre}\ d\ p)$$

La preuve est quasi immédiate avec la définition récurrente de `integre` *et les propriétés de* `ajoute_g` *et* `add_poly`.
On montre enfin par récurrence sur la taille du polynôme p que

$$\forall c \in \mathcal{D},\ \texttt{raffine\_g}(c.p) = c.\texttt{raffine\_g}\ p$$

Là encore, c'est immédiat avec les propriétés de `premier_g` *et* `div2_poly`.
La propriété similaire s'en déduit pour `raffine_d` avec les propriétés de `inverse_ldb`.

Q.24. On a

$$I(p) = I(\langle 1; -2; 4\rangle) = 1 + 2X - 3X^2 = (1-X)(X+1/3)$$

qui est positif sur $[0,1]$. Avec la propriété admise et la question précédente, on aura

$$\forall k \in \mathbb{N},\ \phi^k(p) = \frac{1}{16^k}p \ \text{ avec } \ \phi\ :\ q \mapsto \texttt{raffine\_d}(\texttt{raffine\_g}\ p)$$

Ainsi, l'un des tests profondeur $2k$ sera toujours faux (on tombe sur un polynôme dont l'un des coefficienst est < 0) alors que le polynôme p vérifie la bonne propriété de positivité sur $[0,1]$.