Corrigé du DS4

Listes à deux bouts de nombres dyadiques

Partie A – Points faciles sur les listes

A.1

```
# let rec conc l l' = match l with
        | [] -> l'
        | a :: q -> a :: (conc q l');;
conc : 'a list -> 'a list -> 'a list = <fun>
```

A.2 On utilise un accumulateur et la récursivité terminale pour obtenir une complexité linéaire (l'idée est de déverser une pile d'assiettes sur un emplacement vide). Le fait de placer l'accumulateur en première position n'a pas grande importance, cela permet juste d'écrire la fonction miroir en fonction de deverse de façon plus concise.

```
# let miroir =
        let rec deverse l = function
                | [] -> l
                | a :: q -> deverse (a :: l) q
        in
                deverse [];;
miroir : '_a list -> '_a list = <fun>
```

Partie B – Grands entiers

B.1 Il faut bien faire attention à garantir l'invariant lorsqu'on rajoute 0 à [].

```
# let cons_nat c n = if c <> 0 or n <> [] then c :: n else n;;
cons_nat : int -> int list -> int list = <fun>
```

B.2 On utilise une programmation récursive, on compare les termes de plus grand poids, puis, en cas d'égalité, les chiffres des unités.

```
# let rec cmp_nat n1 n2 = match (n1, n2) with
        | ([], []) -> 0
        | ([], _) -> -1
        | (_, []) -> 1
        | (a1 :: q1, a2 :: q2) -> let b = cmp_nat q1 q2 in match b with
                | 0 -> if a1 < a2 then -1 else if a1 > a2 then 1 else 0
                | _ -> b;;
cmp_nat : 'a list -> 'a list -> int = <fun>
```

B.3 On suit les recommandations de l'énoncé.

```
# let rec add_retenue n retenue = if retenue = 0 then n else match n with
        | [] -> cons_nat retenue []
        | a :: q -> let temp = a + retenue in
                if temp < base then temp :: q else (temp - base) :: add_retenue q 1;;
add_retenue : int list -> int -> int list = <fun>

# let rec add_nat_aux ret n1 n2 = match (n1, n2) with
        | ([], _) -> add_retenue n2 ret
```

```
        | (_, []) -> add_retenue n1 ret
        | (a1 :: q1, a2 :: q2) -> let temp = a1 + a2 + ret in
                if temp < base
                        then temp :: (add_nat_aux 0 q1 q2)
                        else (temp - base) :: (add_nat_aux 1 q1 q2);;
add_nat_aux : int -> int list -> int list -> int list = <fun>

# let add_nat = add_nat_aux 0;;
add_nat : int list -> int list -> int list = <fun>
```

B.4

Comme la base est paire, le reste de la division de n par 2 sera immédiatement déterminé par son chiffre des unités, fourni par la commande a0 mod 2. Le quotient de la division de a_0 par 2 est lui fourni par a0 / 2.

Commençons par déterminer mathématiquement reste et quotient : si le grand entier n s'écrit $\sum_{k=0}^{m} a_k b^k$, alors

$$n = a_0 + 2 \cdot \left(\frac{b}{2} \sum_{k=1}^{m} a_k b^{k-1} \right)$$

de sorte que le quotient vaut $a_0/2 + \left(\frac{b}{2} \sum_{k=1}^{m} a_k b^{k-1}\right)$ si a_0 est pair, $(a_0 - 1)/2 + \left(\frac{b}{2} \sum_{k=1}^{m} a_k b^{k-1}\right)$ si a_0 est impair.

Cela fournit une programmation récursive simple de cette fonction : si on écrit $\sum_{k=1}^{m} a_k b^k = r' + 2q'$, alors le quotient cherché vaut q' + (a0 + r') / 2 (qui ne donne pas le même résultat que q' + a0 / 2 + r' / 2 dans le cas où a_0 est impair et $r' = 1$).

```
# let rec div2_nat = function
        | [] -> ([], 0)
        | a :: q -> let (quotient, reste) = div2_nat q in
                (cons_nat ((a + base * reste) / 2) quotient, a mod 2);;
div2_nat : int list -> int list * int = <fun>
```

B.5 Question sans difficulté :

```
# let neg_z z = {signe = - z.signe; nat = z.nat};;
neg_z : z -> z = <fun>
```

B.6 Question sans difficulté, où l'on utilise une « fausse » récursivité pour clarifier la programmation :

```
# let rec add_z z1 z2 = match (cmp_nat z1.nat z2.nat) with
        | 0 ->
                if z1.signe = z2.signe
                then {signe = z1.signe; nat = add_nat z1.nat z1.nat}
                else {signe = 1; nat = []}
        | 1 -> let temp =
                if z1.signe = z2.signe
                then
                        add_nat z1.nat z2.nat
                else
                        sous_nat z1.nat z2.nat
        in
                {signe = z1.signe; nat = temp}
        | _ -> add_z z2 z1;;
add_z : z -> z -> z = <fun>
```

B.7 On peut utiliser la récursivité terminale, ou se contenter d'une bête programmation :

```
(* Première version *)
# let rec mul_puiss2_z p z = match p with
        | 0 -> z
        | _ -> let z' = (mul_puiss2_z (p-1) z) in add_z z' z';;
mul_puiss2_z : int -> z -> z = <fun>
```

```
(* Avec récursivité terminale *)
# let rec mul_puiss2_z p z = match p with
        | 0 -> z
        | _ -> (mul_puiss2_z (p-1) (add_z z z));;
mul_puiss2_z : int -> z -> z = <fun>
```

B.8 Question facile à traiter récursivement :

```
# let rec decomp_puiss2 z = match z.nat with
        | [] -> (z, 0)
        | a :: q ->
                if a mod 2 = 1
                then (z, 0)
                else
                        let z' = { signe = z.signe; nat = fst (div2_nat z.nat)} in
                         let (u', p') = decomp_puiss2 z' in
                                (u', p' + 1);;
decomp_puiss2 : z -> z * int = <fun>
```

Partie C – Nombres dyadiques

C.1 Question assez mal traitée, sans doute par manque d'effort de compréhension de la notion de nombre dyadique :

```
# let div2_dya d = {m = d.m; e = d.e - 1};;
div2_dya : dya -> dya = <fun>
```

C.2 Question relativement difficile, dans la mesure on personne n'a bien traité le cas où les exposants étaient égaux :

```
# let rec add_dya d1 d2 = match d1.e <= d2.e with
        | true -> let temp = decomp_puiss2 (add_z d1.m (mul_puiss2_z (d2.e - d1.e) d2.m))
         in {m = fst temp ; e = d1.e + snd temp}
        | _ -> add_dya d2 d1;;
add_dya : dya -> dya -> dya = <fun>
```

C.3 Il suffit d'utiliser la négation et la question précédente :

```
# let sous_dya d1 d2 = add_dya d1 {m = neg_z d2.m; e = d2.e};;
sous_dya : dya -> dya -> dya = <fun>
```

Partie D – Listes à deux bouts

D.1 On peut proposer entre autres :

```
# let ldb_est_vide l = l.lg = 0 && l.ld = 0;;
ldb_est_vide : ldb -> bool = <fun>

# let ldb_est_vide l = conc l.g l.d = [];;
ldb_est_vide : ldb -> bool = <fun>

# let ldb_est_vide l = l = {g = []; lg = 0; d =[]; ld = 0};;
ldb_est_vide : ldb -> bool = <fun>
```

D.2 Question qui a souvent rapporté des points, mais la programmation était maladroite pour certains : si la liste de gauche est vide, alors, par conformité à l'invariant, la liste de droite n'a qu'un élément :

```
# let premier_g l = if l.lg <> 0 then hd (l.g) else hd (l.d);;
premier_g : ldb -> dya = <fun>
```

D.3 Question facile si on a pris le temps de comprendre ce qu'était une LDB :

```
# let inverse_ldb l = {lg = l.ld; g = l.d; ld = l.lg; d = l.g};;
inverse_ldb : ldb -> ldb = <fun>
```

D.4 Encore une question facile si on est rigoureux :

```
# let ajoute_g x l = invariant_ldb {lg = l.lg + 1; g = x :: l.g; ld = l.ld; d = l.d};;
ajoute_g : dya -> ldb -> ldb = <fun>
```

D.5 La difficulté tenait à ce que le terme le plus à gauche pouvait être à droite !

```
# let enleve_g l = if ldb_est_vide l then failwith "erreur" else
                   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};;
enleve_g : ldb -> ldb = <fun>
```

D.6

```
# let ajoute_d x l = inverse_ldb (ajoute_g x (inverse_ldb l));;
ajoute_d : dya -> ldb -> ldb = <fun>

# let enleve_d l = inverse_ldb (enleve_g (inverse_ldb l));;
enleve_d : ldb -> ldb = <fun>
```

D.7 En fait, la réponse varie selon la méthode employée (celle qui était attendue étant celle donnée dans l'énoncé, à la question suivante) :

Observons déjà que les autres opérations que l'application de invariant_ldb ont un coût constant, et donc un coût moyen constant.

Si on procède en déversant un élément tous les quatre ajouts, il y aura application non triviale de invariant_ldb pour tous les indices de la forme $2 + 4k$: il y a M tels indices, où M est de l'ordre de $N/4$, et le coût moyen sera de $(2M + 4M(M + 1)/2)/N$, et donc linéaire en N.

Si on procède plus astucieusement comme l'énoncé l'indique, il y aura application non triviale de invariant_ldb pour tous les indices de la forme $2^{k+1} - 2$. Ceci se montre par récurrence : si le déversement a eu lieu a l'indice $2^{k+1} - 2$, pour lequel les deux listes sont alors de même taille $2^k - 1$, le suivant aura lieu lorsque la taille de la liste de gauche dépassera de deux unités le triple de celle de la liste de droite, *i.e.* pour la taille $3(2^k - 1) + 2 + (2^k - 1) = 4(2^k - 1) + 2 = 2^{k+2} - 2$.

Le nombre M de ces déversement est de l'ordre de $\log(N)$, et le coût total est de $2(2^M - 1) - 2M)$, et donc le coût moyen est constant !

D.8 On créé d'abord une fonction scinde_l qui scinde une liste en deux, et telle que la taille de la seconde liste soit le second argument.

```
# let rec scinde (l, l') = function
        | 0 -> (l, l')
        | p -> scinde (tl l, hd l :: l') (p - 1);;
scinde : 'a list * 'a list -> int -> 'a list * 'a list = <fun>

# let scinde_l l p = let (l1, l2) = scinde (l, []) p in (miroir l2, miroir l1);;
scinde_l : '_a list -> int -> '_a list * '_a list = <fun>

# let rec equilibre p l l' = let (l1, l2) = scinde_l l p in (l1, conc l' l2);;
equilibre : int -> '_a list -> '_a list -> '_a list * '_a list = <fun>

# let rec invariant_ldb l = if l.lg <= l.ld * c + 1 && l.ld <= l.lg * c + 1 then l else
let k = (l.lg + l.ld) / 2 in match l.lg >= l.ld with
        | true -> let (l1, l2) = equilibre (l.lg - k) l.g l.d in
                  {lg = k; g = l1; ld = l.lg + l.ld - k; d = l2}
        | _ -> inverse_ldb (invariant_ldb (inverse_ldb l));;
invariant_ldb : ldb -> ldb = <fun>
```