Arithmétique multiprécision

L'objectif de ce TP est d'étudier une implémentation des entiers naturels de taille arbitraire et des opérations usuelles sur ces entiers (les entiers fournis en standard avec Caml, de type `int`, sont limités en magnitude à $\pm 2^{30} \approx \pm 10^9$ et ne permettent donc pas d'effectuer des calculs sur des nombres de plusieurs dizaines de chiffres). On convient ici de représenter un nombre entier naturel par la liste de ses chiffres en base 10 ordonnés par puissances de 10 croissantes et sans zéros non significatifs, accompagnée de la longueur de cette liste de façon à avoir accès à cette longueur en temps constant.

1) Fonctions utilitaires

Saisir les déclarations suivantes qui seront utilisées au cours du TP :

```
let BASE = 10;;
type nombre = {lg : int; ch : int list};; (* longueur, liste des chiffres *)

(* constantes usuelles *)
let zéro = {lg = 0; ch = []} and un = {lg = 1; ch = [1]} and deux = {lg = 1; ch = [2]};;

(* chiffre des unités, reste du nombre *)
let décompose(a) = if a = zéro then 0,zéro else hd(a.ch), {lg = a.lg - 1; ch = tl(a.ch)};;

(* x = chiffre, a = nombre multiprécision, retourne x + a*BASE *)
let assemble x a = if (a.lg > 0) or (x > 0) then {lg = a.lg + 1; ch = x::a.ch} else zéro;;

(* décalage de n chiffres à droite *)
let rec prefix >> a n = if n = 0 then a else assemble 0 (a >> (n-1));;

(* décalage de n chiffres à gauche *)
let rec prefix << a n =
  if n = 0 then zéro, a
  else let x,y = décompose(a) in let u,v = y << (n-1) in (assemble x u), v
;;

(* conversion int -> nombre *)
let rec nombre_of_int(x) =
  if x < 0 then failwith "nombre négatif"
  else if x = 0 then zéro
  else assemble (x mod BASE) (nombre_of_int(x/BASE))
;;

(* conversion nombre -> int (peut déborder) *)
let rec int_of_nombre(a) =
  if a = zéro then 0
  else let x,y = décompose(a) in x + BASE*(int_of_nombre y)
;;

(* nombre aléatoire de n chiffres *)
let rec random_nombre n =
  if n = 0 then zéro
  else if n = 1 then {lg = 1; ch = [1 + random__int(BASE-1)]}
  else assemble (random__int BASE) (random_nombre (n-1))
;;

(* affichage avec poids fort à gauche *)
let affiche(a) =
  let rec loop(l) = match l with u::v -> loop(v); print_int(u) |[] -> () in
  loop(a.ch); print_newline()
```

```
;;

(* résultat d'une comparaison *)
type cmpres = Supérieur | Egal | Inférieur;;
```

2) Addition, soustraction, comparaison.

La somme et la différence de deux nombres peuvent être calculées chiffre par chiffre en propageant les retenues, comme appris à l'école primaire. Écrire les fonctions :

```
add_c : nombre -> nombre -> int -> nombre (* addition de deux nombres et d'une retenue *)
sub_c : nombre -> nombre -> int -> nombre (* soustraction de deux nombres et d'une retenue *)
```

La fonction `sub_c` devra provoquer une erreur si la différence à calculer est strictement négative. On définira les opérateurs infixes d'addition et de soustraction entre nombres par :

```
let prefix ++ a b = add_c a b 0;;
let prefix -- a b = sub_c a b 0;;
```

Écrire une fonction : `cmp : nombre -> nombre -> cmpres` qui compare les deux nombres fournis en arguments. Cette fonction doit uniquement examiner les chiffres des arguments pour décider du sens de la comparaison ; elle ne peut pas utiliser l'opération de soustraction puisque celle-ci peut déclencher une erreur en cas de résultat négatif.

3) Multiplication.

a) Multiplication d'un nombre par un chiffre.

Écrire une fonction `mul_1 : nombre -> int -> int -> nombre` qui prend en arguments un nombre multiprécision a, un chiffre b et une retenue c et qui retourne le nombre multiprécision $a \times b + c$. On prendra garde à fournir un résultat sans zéro non significatif.

b) Multiplication de deux nombres, premier algorithme.

Si $b = b_0 + \texttt{BASE} \times b_1$ alors $a \times b = a \times b_0 + \texttt{BASE} \times (a \times b_1)$. Utiliser cette propriété pour définir une fonction `mul : nombre -> nombre -> nombre` qui calcule le produit de deux nombres multiprécision. Si a et b sont deux nombres de n chiffres, quelle est la complexité temporelle du calcul de $a \times b$ avec cet algorithme ?

c) Multiplication de deux nombres, algorithme de Knuth.

Soient a, b deux nombres multiprécision à multiplier. L'algorithme de Knuth (aussi appelé algorithme de Karatsuba) consiste à découper a et b en deux parties : $a = a_0 + \texttt{BASE}^n \times a_1$, $b = b_0 + \texttt{BASE}^n \times b_1$ et à calculer récursivement les trois produits : $c_0 = a_0 \times b_0$, $c_2 = a_1 \times b_1$ et $c_1 = (a_0 + a_1) \times (b_0 + b_1)$. On a alors :

$$a \times b = c_0 + \texttt{BASE}^n \times (c_1 - c_0 - c_2) + \texttt{BASE}^{2n} \times c_2.$$

L'entier n est choisi de sorte à couper l'un des deux multiplicandes, a ou b, en deux parties de mêmes longueurs à une unité près ; ainsi la complexité temporelle pour la multiplication de deux nombres de p chiffres est $\Theta(p^\alpha)$ avec $\alpha = \lg(3) \approx 1.58$. Programmer cet algorithme. On écrira une fonction `mulk : nombre -> nombre -> nombre` ayant même interface que `mul`, et qui utilisera `mul` comme cas de base lorsque l'un des opérandes est de longueur inférieure ou égale à un seuil à définir en constante. On prendra `seuil` $= 1$ pour les premiers essais, puis lorsque `mulk` fonctionnera correctement, on déterminera expérimentalement une valeur de `seuil` optimale à l'aide de la fonction de test suivante :

```
(* essai et chronométrage *)
let test(n) =
  let a = random_nombre(n) and b = random_nombre(n) in

  let t = sys__time()      in
  let c = mul a b          in
  let t1= sys__time() -. t in

  let t = sys__time()      in
  let d = mulk a b         in
  let t2= sys__time() -. t in

  (cmp c d), t1, t2
;;
```

La fonction `test` prend en argument une taille n, tire au hasard deux nombres a et b de n chiffres et chronomètre les temps de calcul de $a \times b$ par les algorithmes `mul` et `mulk`. Le résultat fourni est la comparaison des deux produits (ils doivent être égaux !), et les temps de calcul en secondes. Régler `seuil` de sorte que le temps de calcul du produit de deux nombres de 1000 chiffres par `mulk` soit minimal. Penser à revalider les définitions de `mulk` et de `test` à chaque fois que `seuil` est modifié.

On définira, lorsque la valeur de `seuil` sera définitivement fixée, un opérateur infixe de multiplication par la déclaration : `let prefix ** = mulk;;`

4) Division.

a) Division d'un nombre multiprécision par un `int`.

Soit a un nombre multiprécision et b un entier strictement positif représentable par une valeur de type `int`. Écrire une fonction `div_1 : nombre -> int -> int*nombre` telle que `div_1` a b retourne le couple constitué du reste et du quotient de la division euclidienne de a par b. Cette fonction sera utilisée par la suite avec $1 \leqslant b < \texttt{BASE}^2$, donc on pourra supposer qu'un calcul intermédiaire de la forme $x + \texttt{BASE} \times y$ avec $0 \leqslant x < \texttt{BASE}$ et $0 \leqslant y < b$ peut être effectué en `int` sans débordement, compte tenu de la valeur choisie pour `BASE`.

b) Division de deux nombres multiprécision.

Soient a et b deux nombres multiprécision. Il s'agit de calculer le reste r et le quotient q de la division euclidienne de a par b. L'algorithme suivant, de type « diviser pour régner », est dû à Burnikel et Ziegler :

Soit $n = \lfloor(\text{longueur}(b) - 1)/2\rfloor$.
Si $n = 0$ *:*
  Diviser a *par* b *avec l'algorithme* `div_1`.
Sinon, décomposer a *:* $a = a_0 + \texttt{BASE}^n \times a_1$.
Si $a_1 \geqslant b$ *:*
  Diviser récursivement a_1 *par* b *:* $a_1 = r_1 + b \times q_1$.
  Diviser récursivement $a_0 + \texttt{BASE}^n \times r_1$ *par* b *:* $a_0 + \texttt{BASE}^n \times r_1 = r_0 + b \times q_0$.
  Alors $r = r_0$ *et* $q = q_0 + \texttt{BASE}^n \times q_1$.
Si $a_1 < b$ *:*
  Décomposer b *:* $b = b_0 + \texttt{BASE}^n \times b_1$.
  Diviser récursivement a_1 *par* b_1 *:* $a_1 = r_1 + b_1 \times q_1$.
  Soit $x = a_0 + \texttt{BASE}^n \times r_1 - b_0 \times q_1$.
  Si $x \geqslant 0$ *alors* $r = x$ *et* $q = q_1$, *sinon* $r = x + b$ *et* $q = q_1 - 1$.

Programmer cet algorithme. La correction, la terminaison et l'analyse de sa complexité constituent des exercices intérressants qu'on pourra faire en dehors du TP. On évitera bien sûr de calculer x tel qu'indiqué dans l'algorithme puisqu'on ne dispose pas de soustraction à résultat négatif. L'opérateur de division sera déclaré de manière infixe par : `let rec prefix // a b = ... ;;`

c) Application : calcul approché de $\sqrt{2}$.

On a $(\sqrt{2}-1)(\sqrt{2}+1) = 1$ donc $\sqrt{2} - 1 = \dfrac{1}{2 + (\sqrt{2}-1)} = \dfrac{1}{2 + \dfrac{1}{2 + (\sqrt{2}-1)}} = \ldots = \dfrac{1}{2 + \dfrac{1}{\ddots + \dfrac{1}{2 + (\sqrt{2}-1)}}}$.

Soient p_n, q_n entiers naturels tels que $\dfrac{p_n}{q_n} = \dfrac{1}{2 + \dfrac{1}{\ddots + \dfrac{1}{2}}}$ où n est le nombre de 2. On peut démontrer mathématiquement que $\dfrac{p_n}{q_n} \xrightarrow[n\to\infty]{} \sqrt{2} - 1$ et que deux fractions successives encadrent la limite, ce qui fournit un algorithme de calcul approché de $\sqrt{2}$ à $\texttt{BASE}^{-k}$ près : calculer de proche en proche les fractions $\dfrac{p_n}{q_n}$ jusqu'à en trouver deux consécutives dont la différence est inférieure ou égale à $\texttt{BASE}^{-k}$ puis prendre $\left\lfloor \dfrac{\texttt{BASE}^k \times (p_n + q_n)}{q_n} \right\rfloor$ comme approximation de $\texttt{BASE}^k \times \sqrt{2}$.

Programmer cet algorithme. On cherchera une relation donnant p_{n+1} et q_{n+1} en fonction de p_n et q_n sous la forme $\begin{pmatrix} p_{n+1} \\ q_{n+1} \end{pmatrix} = A \times \begin{pmatrix} p_n \\ q_n \end{pmatrix}$, où A est une matrice convenable. Alors $\begin{pmatrix} p_n \\ q_n \end{pmatrix} = A^n \times \begin{pmatrix} p_0 \\ q_0 \end{pmatrix}$, ce qui permet de calculer p_n et q_n efficacement grâce à l'algorithme d'exponentiation rapide.

5) Complément : racine carrée entière.
Soit $a \in \mathbf{N}$. Il s'agit ici de déterminer $b = \lfloor \sqrt{a} \rfloor$ et $r = a - b^2$.

a) Racine carrée entière d'un `int`.
On montre facilement que le nombre réel $\sqrt{a}$ est point fixe de la fonction $f : x \longmapsto \frac{1}{2}(x + a/x)$ et que la suite (x_n) définie par $x_0 = a$, $x_{n+1} = f(x_n)$ converge vers $\sqrt{a}$ si $a > 0$. Moins facilement, on peut prouver que si $a \in \mathbf{N}^*$ et (x_n) est défini par $x_0 = a$, $x_{n+1} = \lfloor f(x_n) \rfloor$ alors la suite (x_n) (à valeurs entières) est d'abord strictement décroissante puis stationnaire ou oscillante. De plus, le dernier terme de la « partie strictement décroissante » est $b = \lfloor \sqrt{a} \rfloor$.

Écrire une fonction `sqrt_1 : int -> int` qui calcule la racine carrée entière d'un entier naturel de type `int` fourni en argument en exploitant cette propriété. On pensera à traiter à part le cas où l'argument est nul !

b) Racine carrée entière d'un nombre multiprécision.
L'algorithme suivant, qui est une adaptation « diviser pour régner » d'une méthode d'extraction de racine carrée jadis enseignée à l'école, est dû à Zimmermann :

Soit $n = \lfloor (\text{longueur}(a) - 1)/4 \rfloor$.
Si $n = 0$, *calculer* b *avec l'algorithme* `sqrt_1`, *puis* $r = a - b^2$.
Sinon :
  Décomposer a *:* $a = a_0 + \texttt{BASE}^n \times a_1 + \texttt{BASE}^{2n} \times a_2$.
  Calculer récursivement $b_1 = \lfloor \sqrt{a_2} \rfloor$ *et* $r_1 = a_2 - b_1^2$.
  Diviser $a_1 + \texttt{BASE}^n \times r_1$ *par* $2b_1$ *:* $a_1 + \texttt{BASE}^n \times r_1 = r_0 + b_0 \times (2b_1)$.
  Soit $x = a_0 + \texttt{BASE}^n \times r_0 - b_0^2$.
  Si $x \geqslant 0$ *alors* $b = b_0 + \texttt{BASE}^n \times b_1$ *et* $r = x$,
    sinon $b = b_0 - 1 + \texttt{BASE}^n \times b_1$ *et* $r = x + 2(b_0 + \texttt{BASE}^n \times b_1) - 1$.

Programmer cet algorithme. On écrira une fonction `sqrt : nombre -> nombre*nombre` qui retourne le couple (r, b). On dispose ainsi d'un nouvel algorithme de calcul approché de $\sqrt{2}$ à $\texttt{BASE}^{-k}$ près : il suffit de calculer la racine carrée entière de $2 \times \texttt{BASE}^{2k}$. Comparer cet algorithme avec celui déduit du développement en fraction continue de $\sqrt{2} - 1$ vu à la section précédente. On écrira une fonction de test inspirée de celle utilisée pour comparer les algorithmes de multiplication.