ÉCOLE POLYTECHNIQUE

FILIÈRE MP
OPTION INFORMATIQUE

CONCOURS D'ADMISSION 2009

COMPOSITION D'INFORMATIQUE

(Durée : 4 heures)

L'utilisation des calculatrices **n'est pas autorisée** pour cette épreuve.
Le langage de programmation choisi par le candidat doit être spécifié en tête de la copie.

⋆ ⋆ ⋆

Ce polynôme est-il positif ?

Dans certains cas, la démonstration assistée par ordinateur demande de vérifier des inégalités polynomiales de la forme :

$$\forall x \in [0,1],\ x^6 - 42x^5 + 120x^4 - 140x^3 + 75x^2 - 15x + 1 \geq 0$$

Ce problème étudie une technique basée sur les polynômes de Bernstein permettant de démontrer ce style d'inégalités automatiquement.

Les parties **I** et **II** traitent de nombres en précision arbitraire, respectivement des entiers relatifs et des nombres dyadiques. La partie **III** traite de listes que l'on peut manipuler par les deux bouts. Enfin, la partie **IV** introduit les polynômes de Bernstein et traite d'un moyen de démontrer les inégalités.

Les parties peuvent être traitées indépendamment. Mais attention, chaque partie utilise des notations et des fonctions introduites dans les parties précédentes. L'énoncé utilise à plusieurs reprises la formulation « on garantira l'invariant P sur le type τ ». On entend par cela que les fonctions que vous allez écrire peuvent supposer que la propriété P est vraie pour leurs arguments de type τ et qu'en contrepartie elles doivent produire des résultats de type τ vérifiant la propriété P. Si la logique de la fonction conduit à identifier des arguments de type τ « impossibles », le code du candidat peut échouer en appelant la fonction (procédure en Pascal) `echouer` qui prend une (courte) chaîne explicative en argument.

I. Grands entiers

Les nombres que nous allons manipuler nécessitent une précision qui dépasse celle des entiers de la machine (types `int` en Caml et `integer` en Pascal). 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 :

```
(* Caml *)
let base = ...;;
type nat == int list;;
```

```
{ Pascal }
const base : integer = ...;
type nat = ^cellule; cellule = record
valeur : integer; suite : nat; end;
```

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, $0 \leq c <$ `base`) et un grand entier n, et qui renvoie le grand entier $c+$`base`$\times n$. La fonction `cons_nat` peut aider à garantir l'invariant du type `nat`.

```
(* Caml *)  cons_nat : int -> nat -> nat
{ Pascal }  function cons_nat(c : integer; n : 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.

```
(* Caml *)  add_nat : nat -> nat -> nat
{ Pascal }  function add_nat(n1 : nat; n2 : nat) : nat
```

Question 3 Définir une fonction `cmp_nat` qui prend en arguments 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 $n_1 = n_2$.

```
(* Caml *)  cmp_nat : nat -> nat -> int
{ Pascal }  function cmp_nat(n1 : nat; n2 : nat) : integer
```

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

```
(* Caml *)  sous_nat : nat -> nat -> nat
{ Pascal }  function sous_nat(n1 : nat; n2 : 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 de 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.

```
(* Caml *)  div2_nat : nat -> nat * int
{ Pascal }  procedure div2_nat(n : nat; var q : nat; var r : integer)
```

À partir de ces grands entiers naturels, on va maintenant construire de grands entiers relatifs. Pour cela, on introduit le type enregistrement `z` suivant, où le champ `signe` contient le signe de l'entier relatif, à savoir 1 ou -1, et le champ `nat` sa valeur absolue.

```
(* Caml *) type z = { signe: int; nat: nat };;
{ Pascal } type z = record signe: integer; nat: nat; end;
```

On notera que 0 admet deux représentations, ce qui n'est pas gênant par la suite.

Question 6 Définir une fonction `neg_z` qui calcule la négation d'un grand entier relatif.

```
(* Caml *)  neg_z : z -> z
{ Pascal }  function neg_z(z : z) : z
```

Question 7 Définir une fonction `add_z` qui calcule la somme de deux grands entiers relatifs.

```
(* Caml *)  add_z : z -> z -> z
{ Pascal }  function add_z(z1 : z ; z2 ; z) : z
```

Question 8 Définir une fonction `mul_puiss2_z` qui prend en arguments un entier machine p ($p \geq 0$), un grand entier relatif z, et qui renvoie le grand entier relatif $2^p z$. On se contentera d'une solution simple, sans viser particulièrement l'efficacité.

```
(* Caml *)  mul_puiss2_z : int -> z -> z
{ Pascal }  function mul_puiss2_z(p : integer ; z : z) : z
```

Question 9 Définir une fonction `decomp_puiss2_z` qui prend en argument un grand entier relatif z non nul, et qui renvoie un grand entier relatif u impair et un entier machine p tels que $z = 2^p u$. Cette fonction calcule donc la plus grande puissance de 2 qui divise z et renvoie la décomposition correspondante. Comme ci-dessus, on visera la simplicité et on supposera que z est tel que p est bien représentable par un entier machine.

```
(* Caml *)  decomp_puiss2_z : z -> z * int
{ Pascal }  procedure decomp_puiss2_z(z : z ; var u : z ; var p : integer)
```

II. Nombres dyadiques

Un nombre dyadique est un nombre rationnel qui peut s'écrire sous la forme

$$a \times 2^b \quad \text{avec } a, b \in \mathbb{Z}.$$

On note $\mathcal{D}$ l'ensemble des nombres dyadiques. On définit le type `dya` suivant pour représenter les nombres dyadiques :

```
(* Caml *)
type dya = { m : z; e : int };;
```

```
{ Pascal }
type dya = record m : z; e : integer;
end;
```

Si d est une valeur du type `dya`, on l'interprète donc comme le nombre rationnel $d.\mathtt{m} \times 2^{d.\mathtt{e}}$, où $d.\mathtt{m}$ est parfois appelé *mantisse*, et $d.\mathtt{e}$ *exposant*.

On garantira l'invariant suivant sur le type `dya` : la valeur du champ `m` est soit nulle, soit impaire. On supposera par ailleurs que la capacité des entiers machines ne sera jamais dépassée dans le calcul des exposants.

Question 10 Définir une fonction `div2_dya` qui divise un nombre dyadique par 2.

```
(* Caml *)  div2_dya : dya -> dya
{ Pascal }  function div2_dya(d : dya) : dya
```

Question 11 Définir une fonction `add_dya` qui calcule la somme de deux nombres dyadiques.

```
(* Caml *)  add_dya : dya -> dya -> dya
{ Pascal }  function add_dya(d1 : dya; d2; dya) : dya
```

Question 12 Définir une fonction `sous_dya` qui calcule la différence de deux nombres dyadiques.

```
(* Caml *)  sous_dya : dya -> dya -> dya
{ Pascal }  function sous_dya(d1 : dya; d2; dya) : dya
```

III. Listes à deux bouts

On considère maintenant des listes de nombres dyadiques. Si une telle liste contient les n éléments $x_1, x_2, \ldots, x_n$, dans cet ordre, on la note $\langle x_1; x_2; \ldots; x_n \rangle$. Dans la partie IV, nous aurons besoin de manipuler de telles listes aux deux extrémités, c'est-à-dire d'ajouter et de supprimer des éléments à gauche comme à droite, et également de calculer efficacement l'image miroir d'une telle liste, c'est-à-dire la liste $\langle x_n; \ldots; x_2; x_1 \rangle$. La notion usuelle de liste se prêtant mal à de telles opérations (seule la manipulation de l'extrémité gauche de la liste est aisée), l'objectif de cette partie est de réaliser une structure de données raisonnablement efficace pour représenter une telle « liste à deux bouts ». Pour éviter les confusions, nous utiliserons dorénavant le terme de « LDB » pour désigner une liste à deux bouts, et nous continuerons d'utiliser le terme « liste » pour désigner une liste usuelle (le type `list` de Caml ou une liste chaînée traditionnelle de Pascal).

L'idée est d'utiliser non pas une liste mais deux pour représenter une LDB, la première liste représentant la partie gauche de la LDB et la seconde liste sa partie droite. Ainsi l'ensemble des deux listes $g = [1; 2]$ et $d = [5; 4; 3]$ représentera la LDB $\langle 1; 2; 3; 4; 5 \rangle$. La liste g contient les premiers éléments de la LDB, dans le bon ordre, et la tête de cette liste coïncide donc avec l'extrémité gauche

de la LDB ; symétriquement, la liste d contient les derniers éléments de la LDB, en ordre inverse, et la tête de cette liste coïncide donc avec l'extrémité droite de la LDB.

On définit le type `ldb` suivant pour représenter les LDB :

```
(* Caml *)
type ldb = {
lg : int ; g : dya list ;
ld : int ; d : dya list } ; ;
```

```
{ Pascal }
type liste_dya = ^cell_ld ; cell_ld = record
dya : dya ; suite_ld : liste_dya ; end ;
type ldb = record
lg : integer ; g : liste_dya ;
ld : integer ; d : liste_dya ; end ;
```

On se donne une constante entière $c \geq 2$ et on impose sur le type `ldb` les deux invariants suivants :

Le champ `lg` contient la longueur de la liste `g`, et le champ `ld` celle de la liste `d`. (1)

$$\texttt{lg} \leq c \times \texttt{ld} + 1 \quad \text{et} \quad \texttt{ld} \leq c \times \texttt{lg} + 1 \tag{2}$$

Toutes les questions de cette partie garantiront les invariants au sens précisé dans l'introduction du problème, la question 16 étant un peu particulière.

Question 13 Définir une fonction `ldb_est_vide` qui détermine si une LDB est vide.

```
(* Caml *)  ldb_est_vide : ldb -> bool
{ Pascal }  function ldb_est_vide(l : ldb) : boolean
```

Question 14 Définir une fonction `premier_g` qui renvoie l'élément le plus à gauche d'une LDB, *i.e.* telle que `premier_g` $\langle x_1; x_2; \ldots; x_n \rangle = x_1$. On supposera que la LDB contient au moins un élément.

```
(* Caml *)  premier_g : ldb -> dya
{ Pascal }  function premier_g(l : ldb) : dya
```

Question 15 Définir une fonction `inverse_ldb` qui inverse l'ordre des éléments d'une LDB, *i.e.* telle que `inverse_ldb` $\langle x_1; x_2; \ldots; x_n \rangle = \langle x_n; \ldots; x_2; x_1 \rangle$.

```
(* Caml *)  inverse_ldb : ldb -> ldb
{ Pascal }  function inverse_ldb(l : ldb) : ldb
```

Question 16 Définir une fonction `invariant_ldb` qui vérifie si une LDB satisfait bien l'invariant (2) et le rétablit si ce n'est pas le cas. Plus précisément, la fonction `invariant_ldb` renvoie son argument inchangé lorsqu'il vérifie l'invariant et, dans le cas contraire, renvoie une LDB de même contenu vérifiant l'invariant. Attention, dans ce dernier cas, on demande un coût de l'ordre de la taille de la LDB. Indication : pour une LDB contenant ℓ éléments, la répartition qui range les $\lfloor \ell/2 \rfloor$ premiers éléments dans la liste `g` satisfait (2). Enfin les candidats pourront utiliser, sans les définir,

une fonction `concatener` qui concatène deux listes, ainsi qu'une fonction `inverser` qui inverse une liste.

$$\begin{aligned} \texttt{concatener}([x_1;\cdots;x_n],[y_1;\cdots;y_m]) &= [x_1;\cdots;x_n;y_1;\cdots;y_m] \\ \texttt{inverser}([x_1;x_2;\cdots;x_n]) &= [x_n;\cdots;x_2;x_1] \end{aligned}$$

```
(* Caml *)  invariant_ldb : ldb -> ldb
{ Pascal }  function invariant_ldb(l : ldb) : ldb
```

Question 17 Définir une fonction `ajoute_g` qui ajoute un élément à gauche d'une LDB, *i.e.* telle que `ajoute_g` $x\ \langle x_1;x_2;\ldots;x_n\rangle = \langle x;x_1;x_2;\ldots;x_n\rangle$.

```
(* Caml *)  ajoute_g : dya -> ldb -> ldb
{ Pascal }  function ajoute_g(d : dya ; l : ldb) : ldb
```

Question 18 Définir une fonction `enleve_g` qui supprime l'élément le plus à gauche dans une LDB, *i.e.* telle que `enleve_g` $\langle x_1;x_2;\ldots;x_n\rangle = \langle x_2;\ldots;x_n\rangle$. On supposera que la LDB contient au moins un élément.

```
(* Caml *)  enleve_g : ldb -> ldb
{ Pascal }  function enleve_g(l : ldb) : ldb
```

On supposera avoir écrit les fonctions symétriques opérant sur l'extrémité droite de la LDB : `ajoute_d` pour ajouter un élément à droite, `premier_d` pour obtenir l'élément le plus à droite, et `enleve_d` pour supprimer l'élément le plus à droite.

Question 19 Dans cette question, on suppose $c = 3$. On considère une LDB de longueur N obtenue en appliquant successivement N opérations `ajoute_g` à partir d'une LDB vide. Quel est le coût *moyen* de chaque opération `ajoute_g` ? On supposera que le coût de l'opération `invariant_ldb` est constant lorsque la LDB vérifie l'invariant (2) et proportionnel à la longueur de la LDB lorsque celle-ci est réarrangée.

IV. Polynômes de Bernstein

On considère les polynômes B_i^k définis, pour $i,k \in \mathbb{Z}$, par l'ensemble suivant d'équations récursives :

$$\begin{cases} B_0^0 &= 1 \\ B_i^k &= (1-X)B_i^{k-1} + XB_{i-1}^{k-1} \quad \text{si } 0 \le i \le k \\ B_i^k &= 0 \quad \text{si } i < 0 \text{ ou } i > k \end{cases}$$

On note que B_i^k est non nul si et seulement si $0 \le i \le k$ et que les B_i^k sont des polynômes à coefficients entiers.

Étant donnée une séquence de $k+1$ nombres dyadiques $p = \langle d_0; \ldots; d_k \rangle$, on l'interprète comme un polynôme à coefficients dyadiques, noté $I(p)$, de la manière suivante :

$$I(\langle d_0; \ldots; d_k \rangle) \stackrel{\text{def}}{=} \sum_{i=0}^{k} d_i B_i^k$$

Dans la suite de ce problème, on ne s'intéresse qu'à des polynômes qui s'écrivent sous cette forme. En particulier, quand on écrira « le polynôme p » on signifiera implicitement que p est une LDB de nombres dyadiques, de type `ldb`, interprétée comme le polynôme $I(p)$. On introduit donc le type `poly` suivant pour les polynômes, comme un synonyme pour le type `ldb` des listes à deux bouts de nombres dyadiques :

```
(* Caml *) type poly == ldb;;
{ Pascal } type poly = ldb;
```

Pour un polynôme $p = \langle d_0; \ldots; d_k \rangle$, on appelle k sa *taille* ; le polynôme vide a la taille -1 par convention. On suppose avoir écrit deux fonctions `add_poly` et `sous_poly` calculant respectivement la somme et la différence de deux polynômes de même taille, ainsi qu'une fonction `div2_poly` multipliant un polynôme par la fraction $1/2$.

Question 20 Soit $p = \langle d_0; \ldots; d_k \rangle$ un polynôme. Montrer que si, pour tout i, d_i est positif ou nul, alors

$$\forall x \in [0,1],\ I(p)(x) \geq 0$$

Montrer que la réciproque est fausse avec $p = \langle 2; -1; 2 \rangle$.

Pour un polynôme p de taille $k \geq 0$, on définit le polynôme `derive` p de taille $k-1$ par

$$\texttt{derive}\ p \stackrel{\text{def}}{=} \texttt{sous\_poly}\ (\texttt{enleve\_g}\ p)\ (\texttt{enleve\_d}\ p)$$

Pour un polynôme p de taille k quelconque et $c \in \mathcal{D}$, on définit le polynôme `integre` c p de taille $k+1$ par

$$\texttt{integre}\ c\ p \stackrel{\text{def}}{=} \begin{cases} \texttt{ajoute\_g}\ c\ \texttt{ldb\_vide} & \text{si } p \text{ est vide} \\ \texttt{ajoute\_g}\ c\ (\texttt{add\_poly}\ p\ (\texttt{integre}\ c\ (\texttt{enleve\_d}\ p))) & \text{sinon} \end{cases}$$

Où `ldb_vide` représente la LDB vide.

Enfin, pour un polynôme p de taille k quelconque, on définit les polynômes `raffine_g` p et `raffine_d` p, de taille k, par

$$\texttt{raffine\_g}\ p \stackrel{\text{def}}{=} \begin{cases} \texttt{ldb\_vide} & \text{si } p \text{ est vide} \\ \texttt{integre}\ (\texttt{premier\_g}\ p)\ (\texttt{div2\_poly}\ (\texttt{raffine\_g}\ (\texttt{derive}\ p))) & \text{sinon} \end{cases}$$

et

$$\texttt{raffine\_d}\ p \stackrel{\text{def}}{=} \texttt{inverse\_ldb}\ (\texttt{raffine\_g}\ (\texttt{inverse\_ldb}\ p))$$

On admet alors les résultats suivants : pour tout polynôme p, on a

$$\forall x,\ I(\texttt{raffine\_g}\ p)(x) = I(p)(x/2) \tag{3}$$

et

$$\forall x,\ I(\texttt{raffine\_d}\ p)(x) = I(p)(1/2 + x/2) \tag{4}$$

Question 21 Un calcul donne `raffine_g` $\langle 2; -1; 2\rangle = \langle 2; 1/2; 1/2\rangle$ et `raffine_d` $\langle 2; -1; 2\rangle = \langle 1/2; 1/2; 2\rangle$. Que peut-on en conclure ?

Question 22 Définir une fonction `test_pos` qui prend en argument un polynôme p et qui est une procédure de semi-décision pour la propriété $\forall x \in [0,1], I(p)(x) \geq 0$. Une procédure de semi-décision termine toujours ; si elle renvoie vrai, alors la propriété est vraie ; si elle renvoie faux, alors on ne sait rien de la validité de la propriété. La procédure envisagée est du style « diviser pour régner », et on fixera une limite maximale sur la profondeur de décomposition, au delà de laquelle l'effort de preuve est abandonné. Il n'est pas démandé d'écrire le code des fonctions `raffine_g` et `raffine_d`, que l'on pourra donc appeler sans les définir.

```
(* Caml *)  test_pos : poly -> bool
{ Pascal }  function test_pos(p : poly) : boolean
```

Question 23 Montrer que pour tout polynôme p et tout nombre dyadique c on a les égalités suivantes :

$$\begin{aligned} \texttt{raffine\_g}\,(c \cdot p) &= c \cdot \texttt{raffine\_g}\,p \\ \texttt{raffine\_d}\,(c \cdot p) &= c \cdot \texttt{raffine\_d}\,p \end{aligned}$$

Question 24 Montrer que la méthode ci-dessus est incomplète, c'est-à-dire qu'il existe un polynôme p tel que $\forall x \in [0,1], I(p)(x) \geq 0$, et tel que `test_pos` renvoie faux quelle que soit la profondeur de décomposition. Indication : considérer $p = \langle 1; -2; 4\rangle$, et admettre sans démonstration que `raffine_d` (`raffine_g` p) $= 1/16\,p$.

* *
*