La représentation des polynômes et leurs opérations

Le but global de cette ambitieuse série d'exercices est la réalisation d'un programme pour la manipulation des polynômes d'une variable, munis de leurs principales opérations.

1

ANALYSEUR SYNTAXIQUE. Les polynômes qui nous intéressent sont les mots du langage défini par la grammaire non contextuelle suivante :

polynôme → ['–'] *monôme* { ('+' | '–') *monôme* }
monôme → *nombre* '*' *xpuissance* | *xpuissance* | *nombre*
xpuissance → 'X' | 'X' '^' *naturel*
naturel → *chiffre* { *chiffre* }
nombre → *naturel* ['.' { *chiffre* }]
chiffre → '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'

Dans cette grammaire le méta-symbole | exprime la disjonction, les méta-symboles […] encadrent un élément optionnel (c.-à-d. un élément qui peut apparaître ou non), tandis que les méta-symboles {…} encadrent un élément qui peut apparaître un nombre quelconque de fois. Les terminaux sont écrits entre apostrophes et les non-terminaux sont écrits en italiques.

La première règle de cette grammaire dit qu'un polynôme est une suite non vide de monômes séparés, quand il y en a plusieurs, par des signes + et –, le tout pouvant être précédé d'un signe –. Les deux règles suivantes fixent la syntaxe d'un monôme (il y a plusieurs possibilité), les trois dernières définissent les nombres. Par exemple

```
- 4.5 * X^5 + 2 * X^4 + X^3 - X + 123.0
```

est un polynôme correct, montrant plusieurs sortes de monômes permis.

Ecrivez l'analyseur syntaxique correspondant à cette grammaire. On appelle ainsi un programme qui lit un texte (caractère par caractère) et tente d'y reconnaître un mot du langage défini par cette grammaire. La découverte d'une erreur de syntaxe provoque l'affichage d'un message et l'arrêt du programme. Si l'analyseur peut aller jusqu'au bout de la chaîne sans trouver d'erreur, on dit que le texte a été *reconnu*, ou encore qu'il est *syntaxiquement correct*.

2

CODAGE. Un polynôme sera représenté en mémoire par une liste chaînée dont chaque maillon représente un monôme, c'est-à-dire un couple (*coefficient* , *exposant*).

Nous décidons que le polynôme nul sera représenté par la liste vide.

Définissez les structures de données requises et transformez l'analyseur précédent en lui *ajoutant* le code nécessaire pour que, en plus de l'analyse, il produise le codage en mémoire du polynôme reconnu. Nous disons bien des ajouts, rien de ce qui constituait l'analyseur ne doit disparaître.

3

AFFICHAGE. Ecrivez une fonction qui effectue l'affichage d'un polynôme.

4

CODAGE PAR DEGRE DECROISSANT. Modifiez le programme de la question 2 de sorte qu'un polynôme soit représenté désormais par une liste chaînée *rangée par ordre décroissant des degrés* de ses monômes.

5

EVALUATION. Ecrivez une fonction

```
double eval(POINTEUR p, double x);
```

qui calcule et renvoie la valeur du polynôme représenté par **p** pour une valeur **x** de la variable.

6 OPERATIONS SUR LES POLYNOMES. Ecrivez les fonctions

```
POINTEUR plus(POINTEUR a, POINTEUR b);
POINTEUR moins(POINTEUR a, POINTEUR b);
POINTEUR fois(POINTEUR a, POINTEUR b);
POINTEUR quotient(POINTEUR a, POINTEUR b, POINTEUR *reste);
```

qui effectuent les opérations arithmétiques usuelles.

En programmant ces opérations vous vous donnerez les règles de conduite suivantes :

- allouer autant de maillons que vous voudrez ; en particulier, ne pas modifier un polynôme créé (préférer l'allocation d'un maillon nouveau plutôt que la modification d'un maillon existant),
- ne pas s'occuper de la restitution des maillons devenus inutiles (on verra cela à la question suivante).

Pour programmer la multiplication des polynômes on mettra à profit la remarque suivante :

$$P_n \times Q_m = (a_n X^n + P_{n-1}) \times (b_m X^m + Q_{m-1}) = a_n b_m X^{n+m} + a_n X^n \times Q_{m-1} + P_{n-1} \times Q_m$$

7 MEILLEURE GESTION DE LA MEMOIRE. La manière de gérer la mémoire adoptée ici (« allouer sans compter ; ne jamais rendre ») permet l'écriture facile de fonctions **`fois`**, **`plus`**, etc. claires et fiables. Mais, en pratique, une gestion d'une telle prodigalité ne pourra fonctionner longtemps sans un mécanisme de *recyclage de la mémoire* (on dit aussi *garbage collector*), c'est-à-dire un programme qui recherche les blocs de mémoire inutilisés et les remet dans le stock de mémoire libre.

Ce mécanisme doit travailler « en sous-sol », sans polluer les autres objets et algorithmes, qui en bénéficient sans le savoir. Ainsi, nous allons l'ajouter à notre programme sans modifier en quoi que ce soit les fonctions de traitement des polynômes déjà écrites.

Le mécanisme que nous allons réaliser fonctionne de la manière suivante :

- chaque maillon comporte deux champs supplémentaires : un pointeur **`general`** réalisant le chaînage de tous les maillons et un entier **`utile`**, utilisé à un certain moment pour marquer les maillons utiles,
- tous les maillons sont ajoutés à une grande liste au moment de leur allocation ; la variable **`tousLesMaillons`** pointe la tête de cette liste,
- pour que le mécanisme puisse identifier les maillons inutiles il faut donner un accès aux polynômes utiles : un tableau global **`polyUtile`** en contient les adresses.

Dans ces conditions, recycler la mémoire c'est :

- en parcourant les polynômes dont les adresses sont inscrites dans **`polyUtile`**, marquer les maillons utiles (un maillon peut être marqué plusieurs fois, cela n'a aucune importance),
- en parcourant la liste **`tousLesMaillons`**, libérer les maillons non marqués (et, au passage, effacer les marques).

Mettez en place ce mécanisme dans votre programme.

8 VERSIONS RECURSIVES DE L'ADDITION ET LA SOUSTRACTION. Ecrivez des versions récursives des fonctions :

```
POINTEUR plus(POINTEUR a, POINTEUR b);
POINTEUR moins(POINTEUR a, POINTEUR b);
```

☺☹☺☺