UFR de **mathématique** et d'**informatique**
Université de Strasbourg

Listes doublement chaînées

Le but de cette séance de Travaux Pratiques est de représenter des polynômes en utilisant des listes doublement chaînées.

Vous trouverez sur Moodle une archive contenant un certain nombre de fichiers pour vous aider à démarrer. Après l'avoir téléchargée :

```
$ tar -xf sda1-tp3.tar.gz
$ cd sda1-tp3
```

Le fichier `Makefile` contient les règles nécessaires pour compiler le programme intitulé `polynomes`. Ce programme attend que 2 polynômes (1 polynôme par ligne) soient fournis sur l'entrée standard. Pour décrire un polynôme, chaque monôme est composé d'un coefficient et d'un degré sous la forme `<coefficient>*x^<degre>`.

Voici un exemple d'éxécution interactive (d'un programme terminé) :

```
$ ./polynomes
P1(X) = 1.1*x^1 + 3.3*x^3 + 2.2*x^2
P2(X) = 1*x^1 + 4*x^4 + -2*x^2 + 1*x^1 + 1*x^1

P1(X)   = 3.30*X^3 + 2.20*X^2 + 1.10*X^1
P2(X)   = 4.00*X^4 + -2.00*X^2 + 3.00*X^1
P1 + P2 = 4.00*X^4 + 3.30*X^3 + 0.20*X^2 + 4.10*X^1
P1 * P2 = 13.20*X^7 + 8.80*X^6 + -2.20*X^5 + 5.50*X^4 + 4.40*X^3 + 3.30*X^2
P1'(X)  = 9.90*X^2 + 4.40*X^1 + 1.10
P2'(X)  = 16.00*X^3 + -4.00*X^1 + 3.00
```

Pour compléter le programme, il faudra implémenter des listes doublement chaînées de monomes et implémenter les opérations spécifiques aux polynômes. Les seuls fichiers qu'il est nécessaire de modifier sont les fichiers `liste.c` et `polynome.c`. Votre implémentation doit respecter la documentation présente dans les fichiers `liste.h` et `polynome.h`.

Quelques fichiers de test ainsi que les résultats attendus sont disponibles dans les répertoires `tests` et `results`. Par exemple pour un test `X` :

```
$ ./polynomes <tests/X.txt
$ ./polynomes <tests/X.txt >resultat
$ diff results/X.txt resultat
```

Les polynômes seront représentés par des listes de monômes. On vous fournit la structure suivante (voir le fichier `monome.h`) pour représenter un monôme :

```
#define MONOME_DEGRE_TYPE unsigned int
typedef MONOME_DEGRE_TYPE monome_degre_type;

#define MONOME_COEFFICIENT_TYPE double
typedef MONOME_COEFFICIENT_TYPE monome_coefficient_type;

struct monome
{
  /** \brief Degré du monome. */
  monome_degre_type degre;
  /** \brief Coefficient du monome. */
  monome_coefficient_type coefficient;
};
typedef struct monome monome;
```

On vous fournit également les fonctions suivantes (voir le fichier `monome.h`) pour manipuler ces monômes :

```
monome monome_addition(monome, monome);
monome monome_multiplication(monome, monome);
int monome_fscan(FILE*, monome*);
int monome_fprint(FILE*, monome);
int monome_sscan(const char*, monome*);
```

1 Implémentation des listes doublement chaînées

Les listes doublement chaînées (voir Figure 1) permettent un parcours dans les deux sens de la chaîne à partir de n'importe quel chaînon de la chaîne. On peut alors retrouver tous les autres maillons à partir d'un pointeur vers n'importe quel chaînon de la liste.

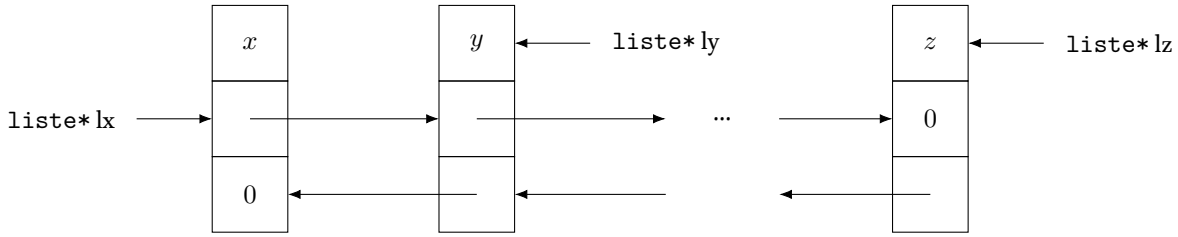


FIGURE 1 – Représentation graphique d'une liste doublement chaînée

Pour ce faire, chaque chaînon doit avoir connaissance, s'ils existent, de son prédécesseur et de son successeur. Voici la structure proposée :

```
struct liste
{
  /** \brief Terme. */
  monome terme;
  /** \brief Élément suivant dans la liste. */
  struct liste* suivant;
  /** \brief Élément précédent dans la liste. */
  struct liste* precedent;
};
```

Ainsi, dans l'exemple en Figure 1, les pointeurs `lx`, `ly` et `lz` peuvent être utilisés indifféremment pour référencer la liste entière. Il suffit ensuite de parcourir la liste dans l'un ou l'autre sens si une opération donnée nécessite de se trouver à une position précise de la liste (début, fin, milieu, etc.).

Dans un premier temps, implémentez les opérations élémentaires sur les listes :

```
liste* liste_vide(void);                    // Créée une liste vide.
liste* liste_inserer(liste* l, monome t);   // Insérer un monôme.
liste* liste_supprimer(liste* l);           // Supprimer le chaînon.
void liste_free(liste* l);                  // Détruit une liste.
monome liste_monome(const liste* l);        // Le monôme du chaînon.
liste* liste_suivant(const liste* l);       // Maillon suivant.
liste* liste_precedent(const liste* l);     // Maillon précédent.
bool liste_est_vide(const liste* l);        // La liste est-elle vide ?
bool liste_est_debut(const liste* l);       // Premier chaînon ?
liste* liste_debut(const liste* l);         // Début de la liste.
liste* liste_ieme(const liste* l, size_t i); // Retourne le i^me chaînon.
size_t liste_longueur(const liste* l);      // Longueur de la liste.
```

Il faudra veiller à ce que les opérations fonctionnent sur n'importe quel chaînon de la chaîne. Certaines opérations commme par exemple `liste_ieme()`, `liste_longueur()` ou `liste_free()` doivent prendre en compte la liste dans son ensemble. À l'inverse que d'autres opérations telles que `liste_suivant()`, `liste_inserer()` ou `liste_supprimer()`, par exemple, s'effectuent directement sur le chaînon reçu en paramètre.

Voici quelques exemples concrets sur les listes :

— Insérer un nouveau terme au milieu d'une liste avec la fonction `liste_inserer()` :

```
/* En supposant qu'on ait une liste foo. */
const size_t moitie = liste_longueur(foo) / 2;
const monome terme = (monome) { .coefficient = 2.5, .degre = 2, };
liste* const milieu = liste_ieme(foo, moitie);
milieu = liste_inserer(milieu, terme);
foo = liste_debut(milieu);
```

— Obtenir le i^{me} chaînon d'une liste à partir d'un pointeur sur un chaînon $j > i$ en utilisant la fonction `liste_ieme()`.

```
/* En supposant qu'on ait une liste foo d'au moins 4 éléménts. */
liste* const troisieme = liste_ieme(foo, 2);
liste* const quatrieme = liste_suivant(troisieme);
liste* const troisieme_bis = liste_ieme(quatrieme, 2);
/* troisieme et troisieme_bis doivent être égaux. */
```

— Obtenir la longueur d'une liste à partir d'un pointeur sur n'importe quel chaînon de la liste en utilisant la fonction `liste_longueur()` :

```
/* En supposant qu'on ait une liste foo. */
const size_t longueur_foo = liste_longueur(foo);
const size_t moitie = longueur_foo / 2;
liste* const milieu = liste_ieme(foo, moitie);
const size_t longueur_milieu = liste_longueur(milieu);
/* longueur_foo et longueur_milieu doivent être égaux. */
```

— Supprimer le i^{me} chaînon avec la fonction `liste_supprimer()` :

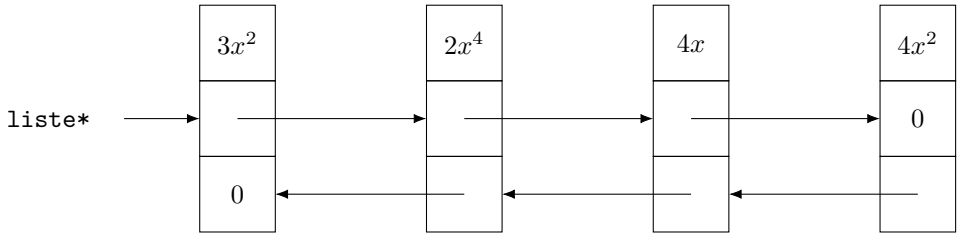
```
/* En supposant qu'on ait une liste foo d'au moins 11 éléments. */
const size_t i = 10;
liste* const ieme = liste_ieme(foo, i);
```

```
4  foo = liste_supprimer(ieme);
5  foo = liste_debut(foo);
```

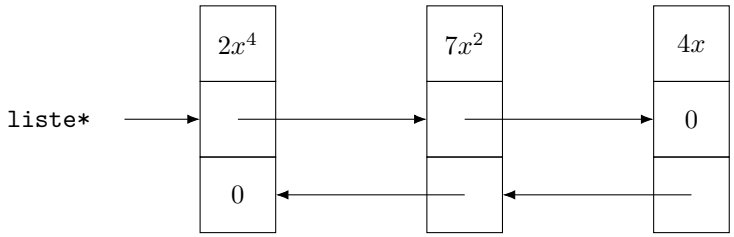
2 Fonctions supplémentaires

Les listes telles qu'elles ont été décrites dans l'exercice précédent peuvent contenir un nombre quelconque de monômes dans un ordre indéterminé. Pour faciliter la manipulation de polynômes, il est préférable de « normaliser » (voir Figure 2) ces listes :

— la liste doit contenir 0 ou 1 monôme pour un degré d donné.
— les monômes doivent être ordonnés par ordre décroissant de degrés.



(a) Liste de monômes non ordonnés :
La liste contient 4 termes non ordonnés dont deux termes de même degré ($3x^2$ et $4x^2$).



(b) Liste de monômes ordonnés :
La liste ne contient plus que 3 termes, ordonnés par ordre décroissant de degrés. Les deux termes de degré 2 ont été additionnés ($3x^2 + 4x^2 = 7x^2$).

FIGURE 2 – Transformation d'une liste non normalisée (Figure 2a) en une liste normalisée (Figure 2b).

Implémentez cette normalisation dans la fonction `liste_normaliser()`. Pour effectuer cette normalisation, vous devrez déplacer des monômes au sein de la liste. Il est fortement conseillé de créer une fonction intermédiaire `liste_echanger()` qui se chargera d'échanger le monôme contenu dans un chaînon avec le monôme contenu dans le chaînon suivant. N'oubliez pas non plus d'additioner les monômes de même degré.

Vous devez donc dans cet exercice implémenter deux nouvelles fonctions :

```
liste* liste_echanger(liste* l);   // Échange le monôme avec le suivant.
liste* liste_normaliser(liste* l); // Normalise une liste de monômes.
```

Astuce : une autre façon de voir `liste_normaliser()` est de la décomposer. Il s'agit de :

— trier les monômes par degrés décroissants
— fusionner les monômes de même degré (additionner les coefficients)

3 Implémentation des polynômes

Une fois « normalisée », une liste de monômes peut être considérée comme un polynôme :

```
typedef liste polynome;
polynome* polynome_addition(const polynome* p1, const polynome* p2);
polynome* polynome_multiplication(const polynome* p1, const polynome* p2);
polynome* polynome_derivee(const polynome* p1);
```

Ainsi, les polynômes sont des listes spéciales de monômes et les fonctions pour les listes peuvent directement être utilisées. En revanche, l'implémentation des fonctions dédiées aux polynômes ne nécessite pas de connaître les détails d'implémentation d'une liste de monômes. Dans cet exercice, **il faut donc manipuler les listes sans jamais accéder directement aux champs de** `struct liste`. Normalement, les fonctions suivantes devraient suffire pour implémenter les opérations demandées ci-dessus :

```
monome monome_addition(monome m1, monome m2);
monome monome_multiplication(monome m1, monome m2);
liste* liste_vide(void);
liste* liste_suivant(const liste* l);
monome liste_monôme(const liste* l);
bool liste_est_vide(const liste* l);
liste* liste_inserer(liste* l, monome t);
liste* liste_normaliser(liste* l);
```

Pour rappel, pour deux polynômes f et g tels que :

$$\begin{aligned} f &= a_n X^n + a_{n-1} X^{n-1} + \ldots + a_1 X + a_0 = \sum_{i=0}^{n} a_i X^i \\ g &= b_n X^n + b_{n-1} X^{n-1} + \ldots + b_1 X + b_0 = \sum_{i=0}^{n} b_i X^i \end{aligned}$$

Les opérations qui vous sont demandées sont :

$$\begin{aligned} f + g &= \sum_{i=0}^{n} (a_i + b_i) X^i \\ f * g &= \sum_{i=0}^{n} \left(a_i X^i \sum_{j=0}^{n} b_j X^j \right) \\ f' &= \sum_{i=0}^{n} a_n \times n X^{n-1} \\ g' &= \sum_{i=0}^{n} b_n \times n X^{n-1} \end{aligned}$$

A Changement de type à la compilation

En observant le code, vous constaterez que le fichier `monome.h` contient des lignes telles que :

```
#ifndef MONOME_COEFFICIENT_TYPE
  #define MONOME_COEFFICIENT_TYPE double
  #ifndef MONOME_COEFFICIENT_SCN
    #define MONOME_COEFFICIENT_SCN "%lf"
  #endif
  #ifndef MONOME_COEFFICIENT_PRI
    #define MONOME_COEFFICIENT_PRI "%.2lf"
  #endif
#endif
```

Ces lignes permettent de changer le type pour les coefficients à la compilation (à vos risques et périls) :

```
$ make -B CPPFLAGS='-DMONOME_COEFFICIENT_TYPE="int" -DMONOME_COEFFICIENT_PRI=\"%d\"
↪  -DMONOME_COEFFICIENT_SCN=\"%d\"'
```

Ou bien simplement changer la manière d'afficher les coefficients (dans cet exemple, 6 chiffres après la virgule au lieu de 2) :

```
$ make -B CPPFLAGS='-DMONOME_COEFFICIENT_PRI=\"%.6lf\"'
```

Notez que la manière dont est écrite le fichier `monome.h` impose que les macros pour la lecture (`MONOME_COEFFICIENT_SCN`) et l'affichage (`MONOME_COEFFICIENT_PRI`) soient définies manuellement dès que le type (`MONOME_COEFFICIENT_TYPE`) est défini manuellement tandis que l'inverse n'est pas requis (on peut définir `MONOME_COEFFICIENT_PRI` sans avoir à manuellement définir `MONOME_COEFFICIENT_TYPE`).

B Fuites mémoires

Pour rappel ou information, vous pouvez vérifier l'absence de fuites mémoires à l'aide d'outils tels que `valgrind(1)` (voir Section B.1) ou bien en activant des options d'instrumentation (voir Section B.2).

B.1 Valgrind

Compilez vos programmes avec les informations de *debugging*, puis éxécutez `valgrind(1)` en passant en paramètre le programme cible et ses options :

```
$ make -B CFLAGS="-Og -ggdb"
$ valgrind ./polynomes <tests/00-vide.txt
$ valgrind --leak-check=full ./polynomes <tests/00-vide.txt
```

B.2 Instrumentation

Compilez vos programmes en ajoutant l'option d'instrumentation `-fsanitize=address` puis éxécutez votre programme de la manière classique :

```
$ make -B CFLAGS="-fsanitize=address" LDFLAGS="-fsanitize=address"
$ ./polynomes <tests/00-vide.txt
```