Examen de TD n°2

La durée de l'examen est de 30 minutes, les notes de cours et de TD sont autorisées. La notation prendra en compte la présentation et la clarté des explications (programmes commentés, dessins explicatifs, ...). L'examen sera noté sur 10 points et le barème est donné uniquement à titre indicatif. Il pourra être modifié lors de la notation finale.

Cet examen concerne l'utilisation de listes pour représenter des ensembles polynômes. Les types utilisés dans cette épreuve sont les suivants :

```
typedef struct s_maillon *p_maillon_t;
typedef p_maillon_t       liste_t;
typedef struct s_maillon {
  int valeur;
  p_maillon_t suivant;
} maillon_t;
typedef liste_t polynome_t;
```

Vous pourrez également utiliser si vous le désirez les fonctions usuelles `nil`,`car` et `cdr`. On rappelle que le `cdr` d'une liste vide est, par convention, la liste vide et que le `car` d'une liste vide est indéfini et arrête le programme.

Le polynôme $a_nX^n + a_{n-1}X^{n-1} + \cdots + a_1X + a_0$ sera représenté par la liste $\{a_0, a_1, \ldots, a_{n-1}, a_n\}$.

▷ **Question 1. (1.5 points)** Écrire une fonction C `monome` dont le prototypage est

```
   polynome_t monome(int degre, int coefficient) ;
```

et telle que `(monome(n,a))` renvoie une liste représentant le monôme aX^n. On supposera que le polynôme nul est de degré strictement négatif et représenté par la liste vide.

Réponse. L'objectif est de construire une liste de la forme $[\underbrace{0, \ldots, 0}_{n}, a]$. Voici l'écriture la plus simple de la fonction `monome`.

```
polynome_t monome(int degre, int coefficient) {
  if (degre<0)  return nil() ;
  if (degre==0) return cons(coefficient,nil());
  return cons(0,monome(degre-1,coefficient));
}
```

Pour ceux qui tiennent vraiment à voir les `malloc` et à l'écrire de façon itérative, voici une autre version.

```
polynome_t monome2(int degre, int coefficient) {
  int d;
  polynome_t res=NULL;
  polynome_t courrant=NULL;

  if (degre<0)  return nil() ;

  res = malloc(sizeof(maillon_t));
  res->valeur = coefficient;
  res->suivant = nil();

  for(d=0;d<degre;d++) {
    courrant = malloc(sizeof(maillon_t));
```

```
    courrant->valeur = 0;
    courrant->suivant = res;
    res = courrant;
  }
  return courrant;
}
```

Elle est beaucoup plus longue à écrire car quand on utilise une structure de données récursive, il est plus naturel d'écrire les fonctions de façon récursive. Néanmoins, cette fonction est beaucoup plus efficace que la précédente car la pile est utilisée de façon intensive et inutile lors des appels récursifs. □

▷ **Question 2. (2 points)** Écrire une fonction C `somme` dont le prototypage est le suivant :

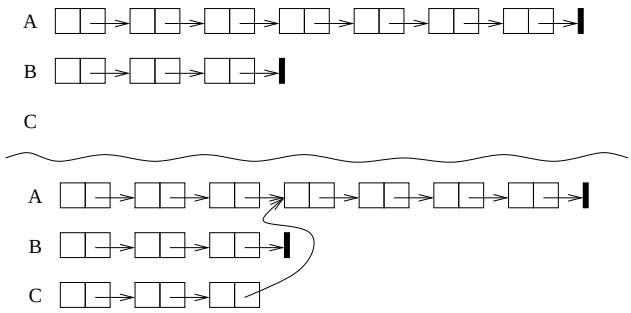
```
   polynome_t somme(polynome_t A, polynome_t B);
```

et qui renvoie la liste correspondant à la somme de `A` et `B`. Vous ferez un dessin explicatif sur le fonctionnement de la fonction et l'état des listes `A`,`B` et `somme(A, B)` avant et après l'appel à `somme`.

Réponse.

```
polynome_t somme(polynome_t A, polynome_t B) {
  if(A==nil()) return B;
  if(B==nil()) return A;
  return cons(car(A)+car(B),somme(cdr(A),cdr(B)));
}
```

Les deux premières lignes sont paresseuses... il y a aliasing entre certains éléments de `C` et certains éléments de `A` ou `B` (voir ci-dessous).



On pourrait éviter ce phénomène en faisant une copie de `A` ou de `B`, par exemple comme ceci :

```
polynome_t somme2(polynome_t A, polynome_t B) {
  if(A==nil()) {
    if(B==nil()) return (nil());
    else return cons(0+car(B), somme(A,cdr(B)));
  }
  if(B==nil()) cons(0+car(A), somme(B,cdr(A)));
  return cons(car(A)+car(B),somme(cdr(A),cdr(B)));
}
```

Si on laisse l'aliasing, il faut faire attention à ne pas modifier `A`, `B` ou `C` sous peine de surprises désagréables (notamment lors du produit qui va être étudié dans les questions suivantes). □

▷ **Question 3. (1.5 points)** Écrire une fonction C `eval` dont le prototypage est le suivant :

```
   int eval(polynome_t A, int x);
```

et qui renvoie la valeur du polynôme A en x en utilisant le schéma d'évaluation de Horner (on rappelle que $\sum a_i x^i = a_0 + x(a_1 + x(a_2 + x(\dots a_{n-1} + x(a_n)\dots)))$).

Réponse.

```
int eval(polynome_t A, int x) {
  if(A==nil()) return 0;
  return (car(A) + x*eval(cdr(A),x));
}
```

□

▷ **Question 4. (2.5 points)** Écrire une fonction C `monome_mult` dont le prototypage est le suivant :

```
   polynome_t monome_mult(int degre,int coefficient, polynome_t A);
```

et qui renvoie la liste représentant le polynôme $aX^i \cdot A[X]$ si `coefficient` vaut a et `degre` vaut i. Vous ferez un dessin explicatif sur le fonctionnement de la fonction et l'état des listes `A`, et `monome_mult(a,i,A)` avant et après l'appel à `monome_mult`.

Réponse.

```
polynome_t monome_mult(int degre,int coefficient, polynome_t A) {
  if((degre<0)||(A==nil())) return nil();
  if(degre==0) return cons(coefficient*car(A),monome_mult(0,coefficient,cdr(A)));
  return cons(0,monome_mult(degre-1, coefficient, A));
}
```

En fait, ici, on ne modifie pas la liste `A`. Une nouvelle liste est créée et il n'y a pas d'aliasing. □

▷ **Question 5. (2.5 points)** Écrire une fonction C `mult` dont le prototypage est le suivant :

```
   polynome_t mult(polynome_t A, polynome_t B);
```

et qui renvoie la liste représentant le polynôme $A \cdot B$.

Réponse.

```
polynome_t mult(polynome_t A, polynome_t B) {
  int degre=0;
  polynome_t C=nil();

  if(A)
    do {
      C=somme(C,monome_mult(degre, car(A), B));
      degre++;
    } while((A=cdr(A)));
  return C;
}
```

On notera cependant que cette solution est très mauvaise en ce qui concerne la gestion de la mémoire puisqu'une liste est créée pour chaque terme de `A` et qu'on ne les libère pas. Pour bien faire, il faudrait se prendre un peu plus la tête... □