# Interpréteur avancé d'expressions arithmétiques

L'objectif est de créer un programme fonctionnel qui met en oeuvre la (presque) totalité des concepts de POO !

Pour cela, il est demandé d'écrire un interpréteur avancé d'expressions algébriques :

1) mettant à la disposition de l'utilisateur des outils de programmation simples pour la création, l'affichage et l'évaluation d'expressions évoluées
2) utilisant des **variables** : le seul type numérique utilisé est le réel (*double*)
3) manipulant des expressions évoluées : les **affectations** dans des variables, les **expressions booléennes**, une **expression *if-then-else***, une **boucle *for*** !..

Les expressions arithmétiques classiques sont :

1) des **constantes numériques de type réel** : 3.7, -0.78 ...
2) des **variables numériques nommées de type réel** : «x», «y» ...
3) des **opérateurs unaires** appliqués à une sous-expression exp - tels que cos(exp) ou sin(exp) - dont le résultat est l'application de l'opérateur sur le résultat de l'évaluation de la sous-expression.
4) des **opérateurs binaires** sur deux opérandes exp1 et exp2 - tel que exp1+exp2 - dont le résultat est l'application de l'opérateur sur les évaluations des deux opérandes.
5) des **affectations** de l'évaluation d'une expression exp dans une **variable** x.
6) des **opérateurs ternaires conditionnels** (cond)? alt1 : alt2 dont le résultat est l'évaluation de la sous-expression alt1 si l'évaluation de la condition cond est vraie (ie. $\neq 0.0$ ou le résultat de l'évaluation de alt2 sinon.
7) Des opérateurs pour le **if then else**, la **boucle for** et un **bloc d'expressions**.

Le principe est de représenter une expression sous la forme d'un arbre dont chaque noeud représente une expression particulière. Les sous-arbres sont les sous-expressions.

L'ensemble des expressions sera représenté par une variable de classe :

**static set<Expression*> _pool;**

On souhaite implémenter chaque expression sous la forme d'une instance d'une classe C++ particulière. On décide donc de créer une classe par type d'expression :

1) une classe Constante, caractérisée par sa valeur réelle non modifiable,
2) une classe pour chaque opérateur unaire, caractérisée par l'opérateur en question et par la sous-expression à laquelle il s'appliquera, par exemple Cos, Sin...
3) *etc*...

# I CONSTANTE ET OPERATEURS UNAIRES

Ecrire une classe constante et les classes Sin, Cos et Exponentielle considérées comme des opérateurs unaires. Ecrire une **fonction de test testUnaires.**

La **méthode de classe toutLiberer** applique itérativement l'operateur delete au premier element du pool tant que le pool n'est pas vide.

```
void testUnaires() {
   Cos *c = new Cos(new Constante(M_PI/3.0));
   cout << *c << " = " << c->eval() << endl;
   Expression::toutLiberer();
}
```

# II OPÉRATEURS BINAIRES

On s'intéresse maintenant aux opérateurs binaires arithmétiques ainsi qu'aux comparaisons. On considérera celles-ci comme des opérateurs binaires dont le résultat de l'évaluation est faux si égal à 0.0 et vrai si différent de 0.0

Ecrire les c**lasses Somme, Difference, Produit, Division, InferieurEgal, SuperieurEgal** qui permettent respectivement de représenter l'addition, la soustraction, la multiplication et la comparaison <= et >= de deux expressions.

Ecrire une **fonction de test testBinaires. Commenter les lignes qui ne sont pas correctes en donnant l'explication adéquate.**

```
void testBinaires() {
   Somme *s = new Somme( new Constante(1.0),
                  new Produit(new Constante(2.0),
                        new Sin(new Constante(M_PI/6.0))
                       ));

   cout << *s << " = " << s->eval() << endl;
   SuperieurEgal comp(s, new Constante(1.8));
   cout << comp << " = " << (bool)comp.eval() << endl;
   Expression::toutLiberer();
}
```

# III VARIABLES

Une même variable peut se retrouver dans plusieurs expressions différentes ou plusieurs fois dans une même expression.
Par exemple l'expression x*cos(x) fait deux fois référence à la variable x.

**L'ensemble des variable est stocké sous la forme d'un tableau associatif : la clé est le nom de la variable stockée et la valeur celle de la variable.**

L'évaluation d'une variable retourne la valeur de la variable. Si la variable n'est pas trouvée dans le tableau associatif alors une nouvelle variable est créee avec la valeur 0.0 qui est retournée.

La **méthode effacerMemoire()** permet d'effacer le contenu de la mémoire (toutes les variables).

L'affectation d'une variable utilise un **opérateur binaire Affectation**.

Ecrire une **classe Variable** et une **classe Affectation** qui permettent d'exécuter le programme de test suivant :

```
void testVariables() {
   Variable x("x", 3.0);
   Variable y("y");
   cout << x << " = " << x.eval() << endl;
   cout << y << " = " << y.eval() << endl;
   Expression *exp = new Somme( new Constante(1.0),
                        new Produit(new Constante(2.0), new Variable("x")));
   Affectation *a = new Affectation(new Variable("y"), exp);
   cout << *a << " = " << a->eval() << endl;
   cout << y << " = " << y.eval() << endl;
   Variable::effacerMemoire();
   cout << y << " = " << y.eval() << endl;
}
```

# IV CONDITIONNEL

Ecrire une **classe Conditionnel** représentant l'**opérateur ternaire (cond) ? e1 : e2.**
Cet opérateur ternaire retourne le résultat de l'évaluation de l'une ou l'autre sous-expression e1 ou e2 en fonction du résultat de l'évaluation d'une expression conditionnelle cond.

La **classe Conditionnel** doit permettre d'exécuter le programme de test suivant :

```
void testConditionnel () {
   Conditionnel *test =
      new Conditionnel(new InferieurEgal(new Variable("x"), new Constante(0.0)),
                  new Cos(new Variable("x")),
                  new Cos(new Produit(new Constante(2.0), new Variable("x"))));
   Variable *x = new Variable("x", M_PI/3.0);
   cout << *x << " = " << x->eval() << endl;
   cout << *test << " = " << test->eval() << endl;
   x->set( -M_PI/3.0);
   cout << *x << " = " << x->eval() << endl;
   cout << *test << " = " << test->eval() << endl;
   Variable::effacerMemoire();
   Expression::toutLiberer();
}
```

# V SI ALORS SINON, BOUCLES ET BLOC

1) **SI ALORS SINON**

   Ecrire une **classe Si_alors_sinon**, dans l'esprit de cet exercice.

   Ecrire une fonction de **test testSi_alors_sinon.**

2) **BOUCLES**

   Ecrire une **classe Pour**, dans l'esprit de cet exercice.

   Ecrire une **fonction de test testPour.**

   Cette fonction affecte la factorielle d'une variable x (dont la valeur a été affectée précédemment) à une variable y, pour x=3.0 puis pour x=5.0.

3) **BLOC**

   Ecrire une **classe Bloc** qui représente une séquence d'instructions (expression) sur le même modèle. On considère que la valeur retournée par un bloc est celle retournée par la dernière instruction/expression du bloc.

   Ecrire une **fonction de test testBloc**.

# VI DERIVATION SYMBOLIQUE

On peut dériver toute expression par rapport à une variable (on utilise le nom de la variable).

La dérivée d'une expression (on ne sait pas laquelle) est une expression (on ne sait pas laquelle)

1) Dérivation d'une constante : on retourne 0
2) Dérivation d'une variable : on retourne 1 si la variable porte le même nom que ce par rapport à quoi on dérive, on retourne 0 sinon
3) Opérateurs unaires : correspond à f(g(x)) : $f$ o $g$

   (f o g)' = g' . (f' o g)

   Exemple : **d cos(x²) /dx = 2x.sin(x²)**
4) Opérateurs binaires : implémenter les relations :

   (f + g)' = f' + g'

   (f - g)' = f'- g'

   (f . g)' = f' g + g' f

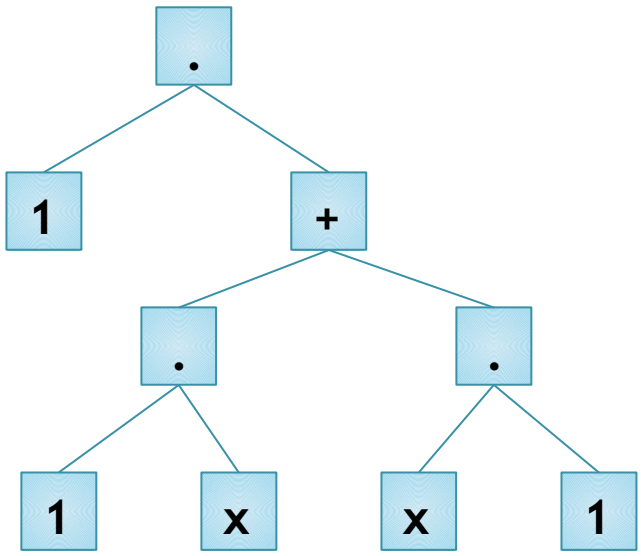
   (f / g)' = (f'g – g'f) / g²

Ecrire le code nécessaire à ces dérivations et une **fonction de test testDerivation.**

# VII SIMPLIFICATION

On peut simplifier une expression pour obtenir une nouvelle expression :

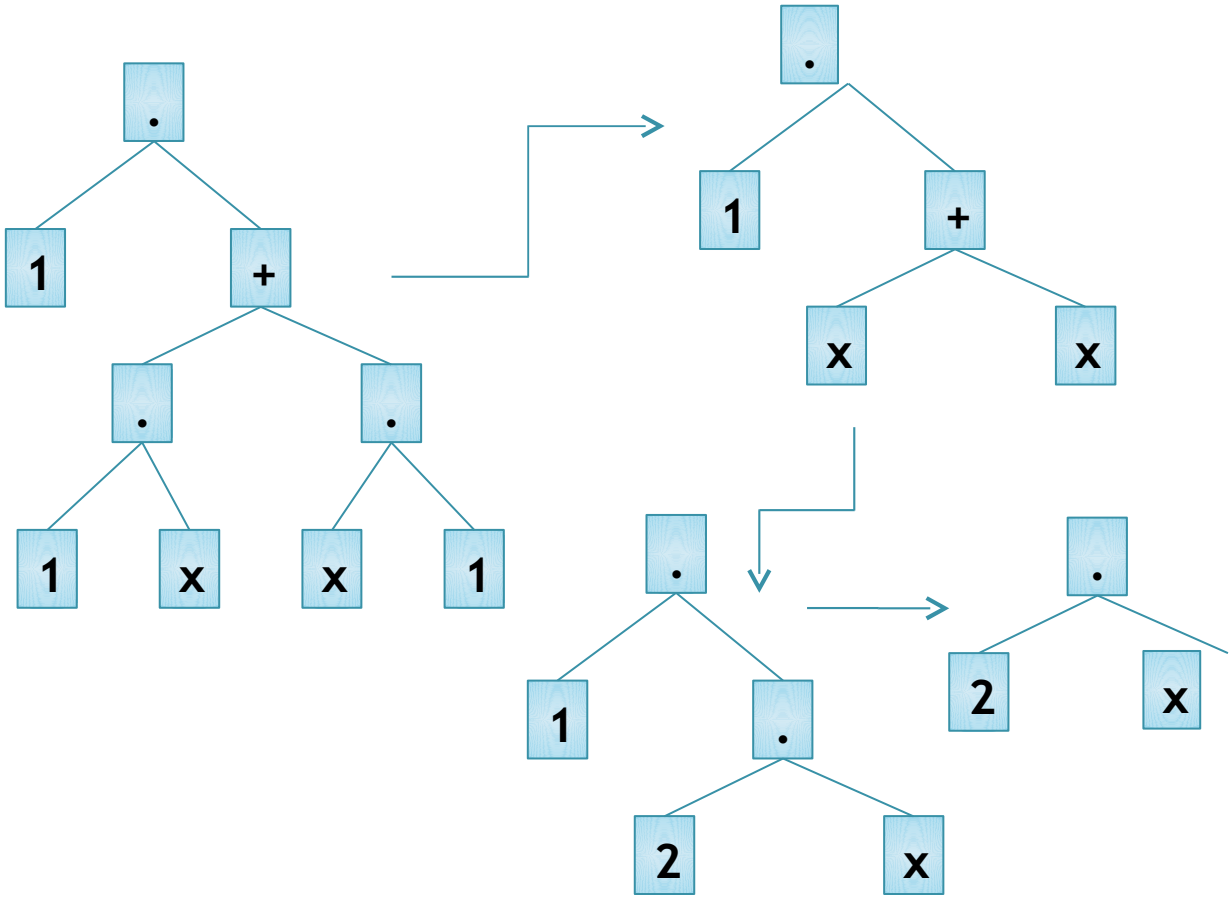
1) Pour les constantes et les variables, on retourne this, car pas de simplification
2) Pour les opérateurs unaires : on ne peut simplifier que l'opérande (il s'agit uniquement des simplifications symboliques pour l'instant).
   On ne peut pas simplifier : cos(0) en 0 par exemple.
3) Pour les opérateurs unaires, il faut simplifier chaque opérande puis regarder si le résultat est simplifiable
   Exemple : 1.((1.x)+(x.1))



**Simplification récursive (la dérivation l'était également)**

Ecrire le code nécessaire à ces dérivations et une fonction de test testDerivation.

Exemple : 1.((1.x)+(x.1))




# IX POLYNÔMES

Ecrire un classe Polynome permettant de représenter des polynômes ordonnés par degré croissant de leurs monômes.

Prévoir des surcharges pour les opérateurs binaires +, -, *, =, +=, -=, *=, ==, !=.

Utiliser la méthode de Horner afin d'évaluer un polynôme P(x) pour une valeur donnée de la variable x.

# ANNEXES

## I set

**std::set<T,...>**

La classe set permet de décrire un ensemble ordonné et sans doublons d'éléments de type T.
Le type T doit disposer d'un constructeur vide T().

Exemple :

```
#include <set>
#include <iostream>
using namespace std;

int main() {
    set<int> s; // équivaut à std::set<int, less<int> >
    s.insert(2); // s contient 2
    s.insert(5); // s contient 2 5
    s.insert(2); // le doublon n'est pas inséré
    s.insert(1); // s contient 1 2 5
    set<int>::const_iterator sit (s.begin());
    set<int>::const_iterator send(s.end());
    for( ; sit != send; ++sit)
        cout << *sit << ' ';
   cout << endl;
   return 0;
}
```

**Attention : le fait de supprimer ou ajouter un élément dans un std::set rend invalide ses iterators. Il ne faut pas modifier un std::set dans une boucle for basée sur ses iterators**

# II map

**std::map<K,T,...>**

Une map permet d'associer une clé (identifiant) à une donnée (table associative).

La map prend au moins deux paramètres templates :

1) le type de la clé K
2) le type de la donnée T

À l'image du std::set, le type K doit être ordonné. Le type T impose juste d'avoir un constructeur vide.

**Attention : le fait de supprimer ou ajouter un élément dans un std::map rend invalide ses iterators. Il ne faut pas modifier un std::map dans une boucle for basée sur ses iterators**.

**Attention : le fait d'accéder à une clé via l'opérateur [ ] insère cette clé (avec la donnée T()) dans la map. Ainsi l'opérateur [ ] n'est pas adapté pour vérifier si une clé est présente dans la map, il faut utiliser la méthode find. De plus, il ne garantit pas la constance de la map (à cause des insertions potentielles) et ne peut donc pas être utilisé sur des const std::map**.

Exemple :

```
#include <map>
#include <string>
#include <iostream>
using namespace std;

int main() {
   map<string, unsigned> map_mois_idx;
   map_mois_idx["janvier"] = 1;
   map_mois_idx["février"] = 2;
   //...
   map<string, unsigned>::const_iterator mit (map_mois_idx.begin());
   map<string,unsigned>::const_iterator mend(map_mois_idx.end());
   for( ;  mit != mend;  ++mit)
      cout << mit->first << '\t' << mit->second << endl;
   return 0;
}
```