Programmation par objets, TP 2: Calcul symbolique

3 séances de TP: 3e, 4e et 5e

Le TP peut être préparé seul ou en binôme. Le rendu doit consister en un fichier archive `<Nom1>_<Nom2>.tar.gz` ou `<Nom1>_<Nom2>.zip` contenant :

- votre code C++, commenté et facilement compilable (commande `make`)
- si la commande `make` ne suffit pas à compiler votre code, ajoutez un fichier README expliquant précisément comment compiler votre code.
- (optionnel) un compte-rendu de TP répondant aux questions qui ne demandent pas un programme ainsi que toute explication complémentaire.

Le TP doit être déposé sur moodle.
La date limite de rendu est la veille de la 6e séance de TP, à 23h59.

On souhaite manipuler de façon symbolique toute expression mathématique de la forme

$$2x^2 + 3 \tag{1}$$

$$2x^2 + \frac{3y - 5xy + 1}{1 + \frac{3}{x}} \tag{2}$$

$$\sin(2\pi(x+1))e^{\frac{1}{x^2}} - \ln(x+5) \tag{3}$$

$$\sqrt{a + \frac{1+y}{1+x}} \tag{4}$$

Contrairement à une calculatrice, le but ici n'est pas de calculer directement le résultat d'une expression, mais de la manipuler (la simplifier, la dériver par rapport à une variable etc.), et éventuellement d'en évaluer une valeur approchée lorsque cela est possible.

Une expression symbolique est composée de différents type d'objets :

les constantes : représentant tout nombre rationnel. On se limitera au type `int` pour le numérateur et le dénominateur pour faire simple.

les constantes symboliques : représentant des nombres irrationnels : π, e etc. A la différence des variables, elle ont un rôle propre, permettant par exemple des simplifications comme $\cos(\pi) = -1, \ln(e) = 1$

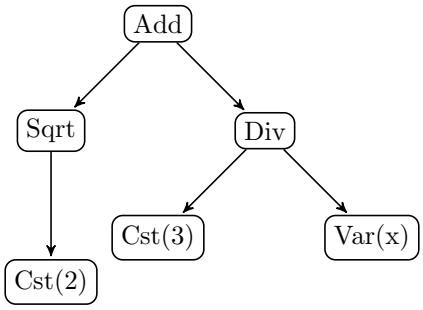
les variables symboliques : toute expression littérale comme a, x, y identifiée par un caractère.

les opérations binaires : $+, -, \times, /$

l'opération unaire : $-$

les fonctions : cos, $\sqrt{}$, ln etc.

Ces objets sont organisé sous la forme d'un arbre pour former une expression.
Ainsi l'expression $\sqrt{2} + \frac{3}{x}$ sera représentée par l'arbre suivant :



Livrable 1. Arbres d'expressions symboliques

On souhaite représenter toute expression symbolique, comme celles listées en introduction.

a. Proposer une classe `Expr` implémentant une expression comme un arbre binaire. Nous avons besoin de connaître le type de chaque noeud, par exemple la simplification demande d'inspecter le type des membres gauche et droit d'une addition.

Le type de chaque noeud est indiqué par un champ `Nature_t nature` où vous définirez `Nature_t` comme un type énuméré :

```
enum Nature_t {CstInt, CstSymb, Var, Add, Sub, Mul, Div, Sqrt, Neg, Ln};
```

b. Implémenter des constructeurs, une methode `affiche`, une méthode `simplifie` de telle sorte que le code

```
    Expr x ('x');
    Expr deux (2);
    Expr neuf (9);
    Expr sqneuf (Sqrt, neuf);
    Expr m (Mul, deux, sqneuf);
    Expr a (Add, x, m);
    a.affiche ();
    cout << " = ";
    a.simplifie ();
    a.affiche ();
    cout << endl;
```

affiche

```
(x + 2 * sqrt (9)) = (x + 6)
```

c. Ajouter d'autres fonctions, de votre choix (par exemple sin, cos, ln, exp, la fonction puissance $(x, n) \longrightarrow x^n$) ainsi que les constantes symboliques π, e.

d. On souhaite maintenant calculer la dérivée d'une expression par rapport à une variable symbolique. Ajouter une méthode `void derive(const Expr v)` prenant un seul argument représentant la variable selon laquelle faire la dérivation et retournant l'expression dérivée correspondante.

e. Vérifier votre code sur plusieurs instances tests.

Pour aller plus loin : substitution et évaluation

f. Implémenter une méthode `subs (const Expr* var, const Expr* exp)`, qui modifie l'expression courante en remplaçant toutes les occurrences de la variable symbolique `var` par l'expression `exp`.

g. Implémenter une méthode `evalue` qui substitue aux constantes entières leurs valeurs flottantes respectives, et effectue les opérations arithmétiques de base.

h. Proposer des implantations de cette méthode `evalue` pour chaque type de noeud classe, y compris les fonctions, en s'appuyant soit sur la bibliothèque standard `math.h`, soit en déduisant un schéma d'approximation du développement limité de ces fonctions. Ainsi toute expression symbolique dont toutes les variables ont été substituées par des expressions sans variable peuvent être évaluées.

i. Proposer plusieurs exemples d'utilisation démontrant la fonctionnalité de votre programme.

Livrable 2. Expressions arithmétiques par polymorphisme

On souhaite représenter par une classe distincte chaque noeud possible permettant de composer des expressions.

On se limite ici aux expressions purement arithmétiques, c'est à dire, sans variable symbolique, ni fonction, et avec uniquement des constantes. Comme par exemple

$$(9+2) \times \left(3 - \frac{1}{5+12}\right). \tag{5}$$

a. Écrire une classe `Cst` (rationnel : numérateur et dénominateur de type `int`).

b. Écrire une classe `Add` et une classe `Mul` qui représentent ces noeuds. Comment faire en sorte que leurs deux membres `gauche` et `droit` puissent être indifféremment de type `Cst`, `Add` ou `Mul` ? Indication : utiliser le polymorphisme par héritage. On introduira en particulier une classe abstraite `ExprArith`.

c. Écrire une méthode `affiche` dans chacune de ces classes qui permette l'affichage de l'expression correspondant à l'arbre ayant pour racine le noeud courant. S'assurer que le parenthésage soit correct (mieux vaut trop que pas assez).

d. Écrire une méthode `simplifie` dans chacune de ces classes qui retourne la constante correspondant à la simplification de l'expression arithmétique correspondante.

e. Améliorer l'affichage pour le cas où la constante est entière (en évitant d'écrire la fraction).

f. Ajouter les classes manquantes (`Div`, `Neg`, `Sub`) et tester votre code sur plusieurs exemples. Par exemple le code :

```
Cst un(1), trois(3), cinq(5);
Mul s(trois,cinq); Add a(un,s);
a.affiche(); std::cout<<" = "; a.simplifie().affiche(); std::cout<<std::endl;

Cst usix(1,6), udemi(1,2);
Add b=Add(usix,udemi);
b.affiche(); std::cout<<" = "; b.simplifie().affiche(); std::cout<<std::endl;
```

doit afficher

```
(1 + (3 * 5)) = 16
(1/6 + 1/2) = 2/3
```