Manipulation d'expressions formelles

Héritage, Polymorphisme

On souhaite manipuler de façon symbolique toute expression mathématique telle que $2x^2+3$, $2x^2+\frac{3y-5xy+1}{1+\frac{3}{x}}$, $\sin(2\pi(x+1))e^{\frac{1}{x^2}}-\ln(x+5)$, ou $\sqrt{a+\frac{1+y}{1+x}}$. Toute expression littérale, comme a, x, y, est une variable symbolique. Une expression peut être représentée par un arbre où :

- les feuilles sont des constantes ou variables symboliques
- les noeuds intermédiaires représentent une opération $(+; -; \times; /)$ ou une fonction ($\sqrt{}$, sin, la fonction puissance, etc.)

Les opérations et fonctions. On distinguera deux types d'opérations :

- les opérations unaires, comme $-$ ou certaines fonctions (sin, exp, $\sqrt{}$) ;
- les opérations binaires, comme $+$, $-$, $\times$, $/$ ou la fonction puissance.

Les constantes. On distinguera deux types de constantes :

- les constantes rationnelles composées de deux nombres entiers, le numérateur et le dénominateur ;
- les constantes symboliques (comme π, e) permettant de manier des nombres irrationnels de façon exacte (par exemple $sin(\pi)=0, ln(e)=1$).

But du TP. On souhaite représenter par une classe distincte chaque noeud possible permettant de composer des expressions. Ainsi l'expression $1+\frac{2}{3}$ sera représentée par un objet instanciant la classe `Addition` et ayant parmis ses champs des références vers une instance de la classe `Constante` contenant le rationnel $\frac{1}{1}$ et une instance de la classe `Constante` contenant le rationnel $\frac{2}{3}$. De nombreuses classes auront ainsi un comportement (et donc un code) proche. Par exemple, la classe Addition et la classe Soustraction auront beaucoup de code en commun. Le but du TP est donc d'exploiter l'héritage et le polymorphisme pour factoriser le code autant que possible.

La classe abstraite `Aexpr` représente les expressions arithmétiques. Elle possède deux méthode abstraites `simplifie` et `toString` qui devront être implémentées par les classes représentant les opérations et constantes :

- `toString` convertit l'expression en chaîne de caractères ;
- `simplifie` retourne (si possible) l'arbre formé d'un unique noeud de type `Constante` contenant la valeur de l'expression arithmétique (sans approximation).

```java
abstract class Aexpr {

    /** Affiche le contenu de l'expression. */
    public void affiche(){ System.out.println(this); }

    /** Réduit l'expression à un unique noeud racine. */
    abstract public Aexpr simplifie();

    /** Convertit l'expression en chaîne. */
    abstract public String toString();
}
```

Exercice 1. Expressions arithmétiques

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

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

Les constantes rationnelles

a. Écrire une classe `Constante` qui hérite de `Aexpr` et représente les constantes rationnelles.
Note : `Constante` devra donc implémenter les méthodes `toString` et `simplifie`.

Les opérations unaires

b. Écrire une classe *abstraite* `UAexpr`, héritant de `Aexpr`, qui encode le comportement commun aux opérations unaires (une expression unaire est un noeud qui possède un fils unique).

c. Écrire une classe `Neg`, héritant de `UAexpr`, qui représente le $-$ unaire.

Les opérations binaires

d. Écrire une classe *abstraite* `BAexpr`, héritant de `Aexpr`, qui encode le comportement commun aux opérations binaires (une expression binaire est un noeud qui possède deux fils).

e. Écrire les classes `Addition`, `Soustraction`, `Multiplication` et `Division`, héritant de `BAexpr`, représentant respectivement $+$, $-$, $*$ et $/$.

Exercice 2. Expressions symboliques

On souhaite maintenant représenter toute expression symbolique, comme celles listées en introduction. On va donc créer de nouvelles classes pour représenter ces nouveaux noeuds. Notons cependant que la sortie de `simplifie` n'est plus nécessairement une constante rationnelle. Par exemple $(2+1)x+3\times 2$ se simplifie en $3x+6$. Il faudra donc mettre à jour si nécessaire les méthodes `simplifie` dans les classes définies précédemment pour garantir ce bon fonctionnement.

Autres types de feuilles

a. Écrire une classe `Variable`, héritant de `Aexpr`, qui représente une variable symbolique (via une chaîne de caractères).

b. Implémenter des classes pour les constantes symboliques (comme π, e)

Autres opérations

c. Ajouter des classes pour représenter les opérations unaires $\surd$, sin, cos, ln et l'opération binaire puissance $(x, n) \longrightarrow x^n$.

d. Implanter aussi les identités classiques : ln(1)=0, $\ln(e) = 1, \sin(\pi) = \sin(0) = \cos(\pi/2) = 0, \cos(\pi) = \sin(-\pi/2) = -1, \cos(0) = \sin(\pi/2) = 1$.

e. On souhaite maintenant calculer la dérivée d'une expression par rapport à une variable symbolique. Modifier vos classes en leur ajoutant une méthode `derive(Variable var)` prenant un seul argument `var`, représentant la variable selon laquelle faire la dérivation et retournant l'expression dérivée correspondante.

f. Vérifier votre code sur plusieurs instances tests.

Exercice 3. Pour aller plus loin : substitution et évaluation

a. Implémenter une méthode `substitue(Variable var, Aexpr exp)`, qui modifie l'expression courante en remplaçant toutes les occurrence de la variable symbolique `var` par l'expression `exp`.

b. Implémenter des méthodes `evalue` qui substitue aux constantes rationnelles une constante flottante (on définira une telle classe `ConstanteFlottante`).

c. Proposer des implantations de cette méthode `evalue` pour chaque classe, y compris les fonctions, en s'appuyant soit sur la bibliothèque Math de Java, soit mieux, en déduisant de développement limités de ces fonctions, un schéma d'approximation.

Ainsi toute expression symbolique dont toutes les variables ont été substituées par des expressions sans variable peuvent être évaluées.

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