

L'objectif du TD et du TP associé est de définir un ensemble de classes permettant de représenter et de manipuler des expressions arithmétiques.

## 1 Le problème

Les expressions sont constituées :

- d'opérandes constantes ou réelles,
- d'opérateurs binaires : +, −, × et /,
- d'opérateurs n-aires : MAXIMUM, MOYENNE, SOMME, ...

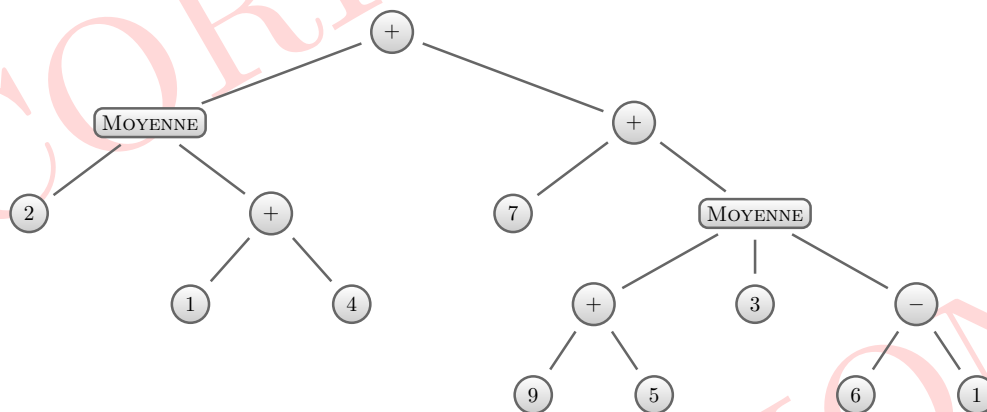
Une expression peut être représentée sous la forme d'un arbre abstrait dont :

- la racine est l'opérateur le moins prioritaire,
- les nœuds (internes) sont des opérateurs,
- les feuilles sont les opérandes.

Par exemple, l'expression :

$$\text{MOYENNE}(2, 1 + 4) + (7 + \text{MOYENNE}(9 + 5, 3, 6 - 1))$$

est représentée par l'arbre abstrait suivant :



## 2 Première approche (sans héritage)

Dans un premier temps, nous allons concevoir une solution sans utiliser l'héritage.

Nous souhaitons pouvoir réaliser deux traitements sur une expression :

- **evaluer** : `expression` → `réel` qui permet d'évaluer la valeur d'une expression.  
Par exemple, l'évaluation de l'expression représentée par l'arbre abstrait présenté précédemment doit calculer la valeur 17.833...
- **decompiler** : `expression` → `chaîne` qui transforme un arbre abstrait en une chaîne de caractères représentant l'expression.  
Ce traitement correspond au traitement inverse qu'il faut réaliser lors de la compilation (où il s'agit de transformer une chaîne de caractères en un arbre abstrait).

▷ **Question 1.** Pour simplifier, on considérera uniquement le cas d'expressions constituées d'opérateurs binaires. Proposer une solution sous la forme d'une seule classe nommée `Expr`.

**Réponse**

On définit une classe `Expr` qui possède :

- 2 attributs `fg` et `fg` désignant les deux opérantes,
- 1 attribut `v` désignant la valeur réelle dans le cas d'une feuille,
- 1 attribut `op` mémorisant le type de l'expression.

```

1 public class Expr {
2     private Expr fg;
3     private Expr fd;
4     private double v;
5     private char op;
6
7     public Expr(double valeur) {
8         this.op = 'v';
9         this.v = valeur;
10    }
11    public Expr(char op, Expr fg, Expr fd) {
12        this.op = op;
13        this.fg = fg;
14        this.fd = fd;
15    }
16
17    public double evaluer() {
18        double resultat = 0.0 ;
19        switch (this.op) {
20            case '+':
21                resultat = fg.evaluer() + fd.evaluer();
22                break;
23            case '-':
24                resultat = fg.evaluer() - fd.evaluer();
25                break;
26            case '*':
27                resultat = fg.evaluer() * fd.evaluer();
28                break;
29            case '/':
30                resultat = fg.evaluer() / fd.evaluer();
31                break;
32            case 'v':
33                resultat = v;
34        }
35        return resultat;
36    }
37
38    public String decompiler() {
39        String resultat = new String();
40        if (this.op == 'v')
41            resultat = Double.toString(this.v);
42        else {
43            resultat += "(";
44            resultat += fg.decompiler();
45
46            resultat += this.op;
47
48            resultat += fd.decompiler();
49            resultat += ")";
50        }
51        return resultat;
52    }
53 }

```

Fin réponse

▷ **Question 2.** Écrire une classe `TestExpr` permettant d'évaluer l'expression  $(9 + 5)/(5 - 3)$  et de "décompiler" l'arbre abstrait que vous avez construit.

Réponse

On commence par dessiner l'arbre abstrait correspondant à l'expression afin de voir quels sont les objets à instancier.

Puis on définit une "vraie" classe de test qui compare les résultats obtenus aux résultats attendus (on peut rappeler qu'une classe de test se doit d'être automatique). Par exemple, la sortie résultant de l'exécution de la classe de test peut ressembler au résultat ci-contre.

```

1 Test evaluer() is ok.
2 Test decompiler() is ok.

```

```

1 public class TestExpr {
2     public static void main(String args[]) {
3         Expr v9 = new Expr(9);
4         Expr v5 = new Expr(5);
5         Expr plus = new Expr('+', v9, v5);
6
7         Expr v5b = new Expr(5);
8         Expr v3 = new Expr(3);
9     }
10 }

```

```

9      Expr moins = new Expr('-', v5b, v3);
10
11      Expr div = new Expr('/', plus, moins);
12
13      double expectedValue = 7.0;
14      if (expectedValue != div.evaluer())
15          System.out.println("Test evaluer() failed.");
16      else
17          System.out.println("Test evaluer() is ok.");
18
19
20      String expectedString = "((9.0+5.0)/(5.0-3.0))";
21      if (expectedString.equals(div.decompiler()))
22          System.out.println("Test decompiler() is ok.");
23      else
24          System.out.println("Test decompiler() failed.");
25  }
26 }

```

Fin réponse

### 3 Seconde approche (héritage et liaison dynamique)

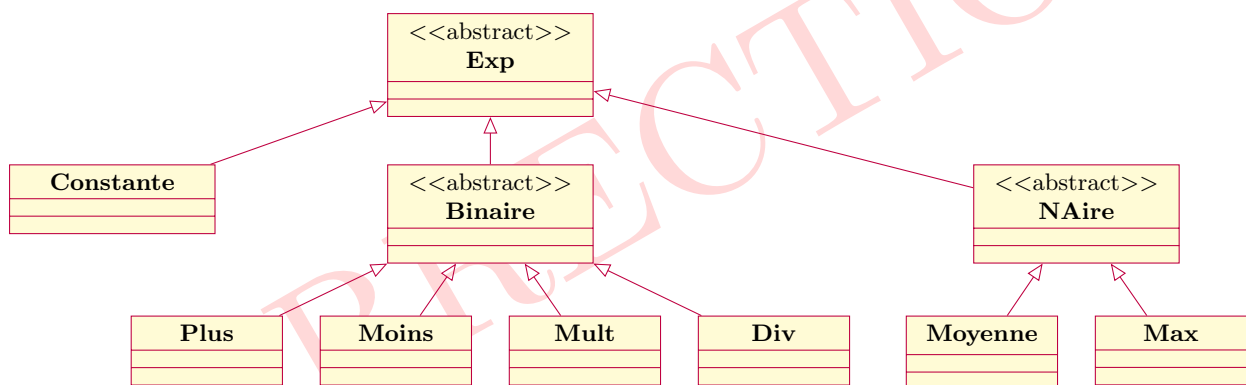
Bien que correcte, l'implémentation proposée dans la section précédente évolue très mal (ajout de nouveaux opérateurs, ajout de nouveaux traitements, ...). Nous souhaitons maintenant utiliser l'héritage et notamment le mécanisme de liaison dynamique afin de proposer une solution relativement aisée à maintenir et bien plus évolutive.

En utilisant l'héritage, il est possible de discriminer les opérations suivant les différentes expressions élémentaires composant une expression arithmétique. De plus, il est également possible de raffiner la classification selon l'arité (le nombre d'opérandes) des opérateurs : constantes, opérateurs binaires, opérateurs n-aires.

▷ **Question 3.** Proposer une hiérarchie de classes permettant de modéliser le problème. Identifier les classes qui seront abstraites.

Réponse

Les classes **Constante**, **Binaire** et **NAire** classifient les opérateurs selon leur arité. La classe **Binaire** factorise le comportement des nœuds binaires et se "décompose" en sous-classes **Plus**, **Moins**, **Mult** et **Div**. Les classes abstraites (\*) servent à factoriser des primitives et à permettre des extensions futures.

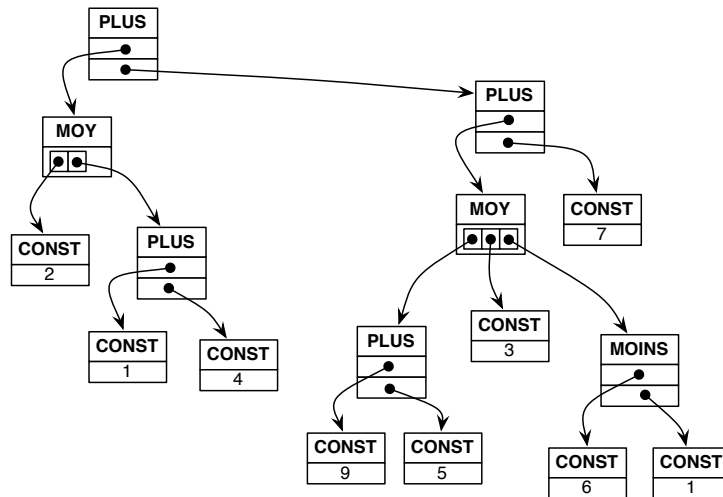


Fin réponse

▷ **Question 4.** Dessiner le schéma mémoire de l'objet qui correspond à l'expression :

MOYENNE(2, 1 + 4) + (7 + MOYENNE(9 + 5, 3, 6 - 1))

Réponse

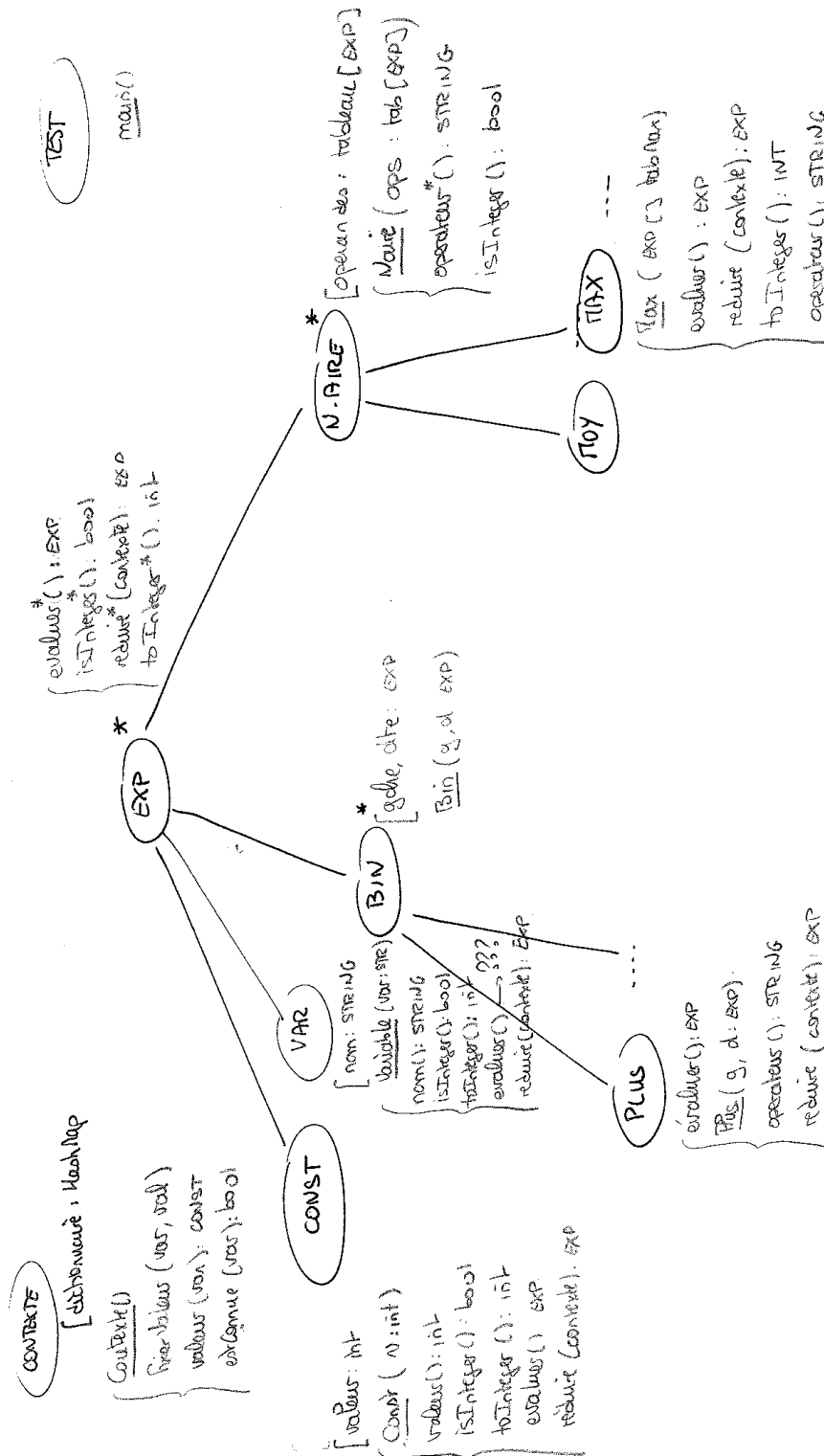


Fin réponse

▷ **Question 5.** Pour réaliser l'évaluation d'une expression, que doit-on ajouter comme primitives (attributs et méthodes) et dans quelles classes ? Étudier le comportement des classes **Exp**, **Binaire**, **NAire**, **Const**, **Plus** et **Moyenne**.

- Dessiner le graphe d'héritage avec les attributs et les méthodes (uniquement leur profil).
- Écrire le corps des différentes méthodes

Réponse



Fin réponse

▷ **Question 6.** On souhaite maintenant étendre nos expressions en autorisant la définition et l'utilisation de variables. Une même expression pourra ainsi être réduite avec différentes valeurs pour ses variables. Par exemple, on définit une variable  $x = 3$  et on souhaite évaluer l'expression  $max(x + 2, 7, 9)$ . Il faut donc être capable de lier le nom de la variable  $x$  avec la valeur 3. Puis, il faut pouvoir remplacer  $x$  par sa valeur dans l'expression. Par exemple, si l'on a la définition  $x = 6$  et l'expression  $max(x + y, y, 9)$ , on peut réduire cette expression en une nouvelle expression  $max(6 + y, y, 9)$ .

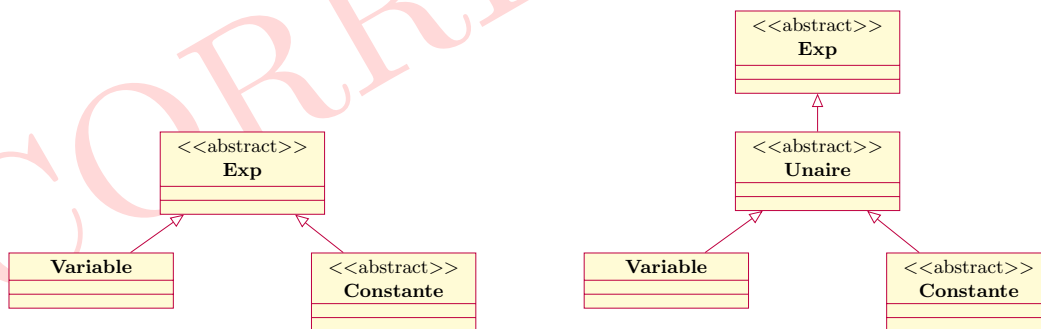
- (a) Définir une nouvelle classe **Variable** qui doit permettre de manipuler des expressions particulières que sont les variables.

- (b) Définir une nouvelle classe **Contexte** permettant de définir le contexte d'évaluation d'une expression en liant une valeur à un nom de variable.
- (c) Pour chaque classe, définir une méthode **Exp reduire(Contexte c)** capable de réduire une expression pour un contexte donné.

Concernant l'expression **Maximum**, on souhaite pouvoir réduire l'expression même partiellement. Par exemple, si la définition  $x = 3$  est présente dans le contexte, la réduction de l'expression  $\max(x + y, y, 7, 9)$  doit donner une nouvelle expression  $\max(3 + y, t, 9)$  (on a évalué la valeur de  $x$  et on a déjà calculé le maximum entre 7 et 9).

### Réponse

- (a) Deux hiérarchies sont envisageables, mais la seconde n'a pas grand intérêt car il est difficile de factoriser quelque chose entre **Variable** et **Constante**. Elle reste cependant intéressante dans le cas où l'on introduirait des opérateurs unaires tels que **ValeurAbsolue** et **Inverse**.



```

1 public class Variable extends Exp {
2     private String nom;
3     public Var(String n) {
4         this.nom = n;
5     }
6 }
  
```

- (b) Il faut noter que la classe **Contexte** ne fait pas partie de l'arbre d'héritage.

Dans cette classe, il faut définir :

- un attribut **memoire** qui permet de stocker le dictionnaire **nom de variable** → **valeur**. Par exemple, pour l'expression  $x = 5$ , il conservera  $x \rightarrow 5$ . (en pratique, on utilisera la classe `java.util.HashMap`).
- un constructeur **Contexte()**.
- une méthode **boolean estConnue(Exp var)** qui retourne vraie si la variable **var** est dans la mémoire.
- une méthode **int valeur(Exp var)** qui retourne la valeur associée à la variable **var**. (la précondition de cette méthode est `estConnue(var) == true`).
- une méthode **fixerValeur(Exp var, int val)** qui associe la valeur **val** à la variable nommée **var** dans la mémoire.

- (c) Voici le pseudo-code de la méthode **reduire()** pour les différentes classes.

```

1 // dans la classe Exp
2 Exp reduire(Contexte contexte)
3     *** méthode abstraite ***
  
```

```

1 // dans classe Variable
2 Exp reduire(Contexte contexte)
3     si (contexte != null et contexte.estConnue(this)) alors
4         retourner contexte.valeur(this) // Attention ici on doit retourner une Exp !
5     sinon
6         retourner this
  
```

```

1 // dans la classe Constante
2 Exp reduire(Contexte contexte)
3     retourner this
  
```

```

1 // dans la classe Binaire
2 Exp reduire(Contexte contexte)
3     Exp gauche = gauche.reduire(contexte)
4     Exp droite = droite.reduire(contexte)
5     si (gauche.estEntier() et droite.estEntier()) alors
6         retourner new Constante(calculer(gauche.versEntier(), droite.versEntier()))
7         // on suppose donc l'existence d'une méthode
8         // versEntier() dans les sous-classes de Binaire
9     sinon
10        // pas de reduction

```

```

1 // dans la classe Plus (et les autres du même genre)
2 la méthode reduire() n'est pas redéfinie. Par contre, on définira une méthode
3 int calculer(int, int) (cf. un peu plus loin dans la correction)

```

```

1 // dans la classe NAire
2 la méthode reduire() n'est pas redéfinie.

```

```

1 // dans la classe Maximum
2 Exp reduire(Contexte contexte)
3     si (estEntier()) alors
4         // le receveur n'est constitué que de constantes entières
5         retourner new Constante(this.versEntier()) // calcul le max des constantes
6     sinon
7         // il n'y a pas que des constantes entières
8         Exp[] ops = new Exp[]
9         int maxi = 0
10        boolean auMoinsUneVar = faux
11
12        pour i:=0 à i=operandes.length
13            Exp elt = operandes[i].reduire(contexte)
14            si (elt.estEntier()) alors
15                int val = elt.versEntier()
16                si (val > maxi)
17                    alors maxi = val
18            sinon
19                ajouter elt à ops
20                auMoinsUneVar = vraie
21        finpour
22
23        si (auMoinsUneVar) alors
24            si (maxi > 0) alors
25                // on crée une Exp constante avec le maxi
26                // et on l'ajoute à la fin de ops qui sera le résultat
27            fsi
28            retourner new Max(op)
29        sinon
30            retourner new Constante(maxi)

```

Il ne faut pas oublier de définir la méthode calculer(), estEntier() et versEntier() dans les différentes classes concernées.

```

1 // dans la classe Exp
2 int versEntier()
3     *** méthode abstraite ***

```

```

1 // dans la classe Binaire
2 int calculer(int gauche, int droite)
3     *** méthode abstraite ***
4
5 int versEntier()
6     si (estEntier()) alors
7         retourner calculer(gauche.versEntier(), droite.versEntier())

```

```

1 // dans les sous-classes de Binaire (Plus, Moins, ...)
2 int calculer(int gauche, int droit)
3     retourner gauche + droit

```

```

1 // dans la classe NAire
2 boolean estEntier()
3     // on boucle sur tous les éléments de operandes
4     // et on teste si operandes[i].estEntier()

```

```

1 // dans la classe Maximum
2 int versEntier()
3     // retourne le maximum des operandes
4     int res = 0

```

```

5   pour i:=0 à nombre d'operandes
6       int valElt = operandes[i].versEntier()
7       si valElt > res alors
8           res = valElt;
9   retourner res

```

Fin réponse

## 4 Préparation et sujet du TP

- ▷ **Question 7.** Afin de préparer le TP de cette semaine,
- (a) Dessiner l'arbre d'héritage des classes
  - (b) Récapituler, pour chaque classe, les attributs et les méthodes.
- ▷ **Question 8.** Programmer les différentes classes

**Remarque :** Pour réaliser le dictionnaire de la classe `Contexte`, vous pouvez vous référer à la documentation<sup>1</sup> de la classe `java.util.HashMap` (constructeurs, méthodes `get()` et `put()`).

- ▷ **Question 9.** Implémenter le processus de “décompilation” en ajoutant les méthodes nécessaires dans les différentes classes. Il s'agit donc de produire une chaîne de caractères dans un buffer de type `java.lang.StringBuffer` (renseignez-vous sur le fonctionnement de la méthode `append()`).

Réponse

*Pour la décompilation, cela occupera ceux qui avancent vite. Sinon je pense que juste coder ce qu'ils ont vu en TD va leur prendre déjà pas mal de temps...*

```

1 // dans la classe Exp
2 String decompiler()
3 // on crée un nouveau StringBuffer buf
4 // on appelle decompilerDans(buf)
5 // on convertira le StringBuffer en un String avant de retourner le résultat
6
7 void decompilerDans(StringBuffer buf)
8     *** méthode abstraite ***

```

```

1 // dans la classe Constante
2 void decompilerDans(StringBuffer buf)
3     buf.append(this.valeur())

```

```

1 // dans la classe Binaire
2 void decompilerDans(StringBuffer buf)
3     buf.append('(')
4     gauche.decompilerDans(buf)
5     buf.append(opérateur())
6     droit.decompilerDans(buf)
7     buf.append(')');
8
9 char opérateur()
10     *** méthode abstraite ***

```

```

1 // dans la classe NAire
2 void decompilerDans(StringBuffer buf)
3     buf.append(opérateur())
4     buf.append('(');
5     pour i de binf à bsup faire
6         operandes[i].decompilerDans(buf)
7         si (i < bsup) alors
8             buf.append(',')
9     buf.append(')');
10
11 char opérateur()
12     *** méthode abstraite ***

```

```

1 // dans les sous-classes de Binaire (Plus, Moins, ...) ou de NAire (Maximum, Moyenne)
2 char opérateur()
3     retourne '+'

```

1. <http://docs.oracle.com/javase/6/docs/api/>



---

**Fin réponse**

CORRECTION

CORRECTION