Programmation Orientée Objet

TP abstraction – Expressiez-vous !

Ensimag 2ème année 2015-16

1 Introduction

Ce TP a pour objectif de manipuler des expressions arithmétiques fonction d'une ou plusieurs variables réelles : affichage, calcul, etc.

La notation usuelle d'une expression est dite *infixée*. Cette notation impose des parenthèses en fonction des priorités des opérateurs utilisés. Voici quelques exemples en notation infixée :

$$E_{infixée}(y) = 3 * y \tag{1}$$

$$F_{infixée}(x) = (x + x) * 5 \tag{2}$$

$$G_{infixée}(a, b) = -3.5 * sin(a + b) \tag{3}$$

Une autre notation est possible, dite *préfixée* - ou encore *polonaise* - dans laquelle un opérateur est toujours placé *devant* les opérandes sur laquelle il s'applique :

$$operateur\ operande1\ operande2 \tag{4}$$

On donne ci-dessous les mêmes expressions `E` `F` et `G`, cette fois ci en notation préfixée. Vérifiez, moyennant un peu de gymnastique intellectuelle, que ce sont bien les mêmes expressions. Notez au passage qu'en notation préfixée il n'est plus nécessaire d'utiliser des parenthèses.

$$E_{prefixée}(y) = *\ 3\ y \tag{5}$$

$$F_{prefixée}(x) = *\ +\ x\ x\ 5 \tag{6}$$

$$G_{prefixée}(a, b) = *\quad -3.5\quad sin\quad +\quad a\quad b \tag{7}$$

Pour représenter une expression arithmétique, on construira en mémoire un arbre *d'objets* Java. L'arbre est une traduction directe de la notation préfixée. Chaque noeud d'un tel arbre représente soit une valeur (*e.g.* -3.5), soit une constante (*e.g* `x` ou `y`), soit un opérateur binaire (*e.g.* `+`), soit un opérateur unaire (*e.g.* `sin`). Les arbres représentant les trois exemples d'expressions sont donnés sur la figure 1.

Dans ce TP, pour simplifier, on ne considèrera que quelques opérateurs (l'extension à d'autres opérateurs ne présentant aucune difficulté). On considèrera donc uniquement :

- des constantes (des valeurs)
- des variables
- les opérateurs binaires `+` et `*`
- les opérateurs unaires `sin cos`

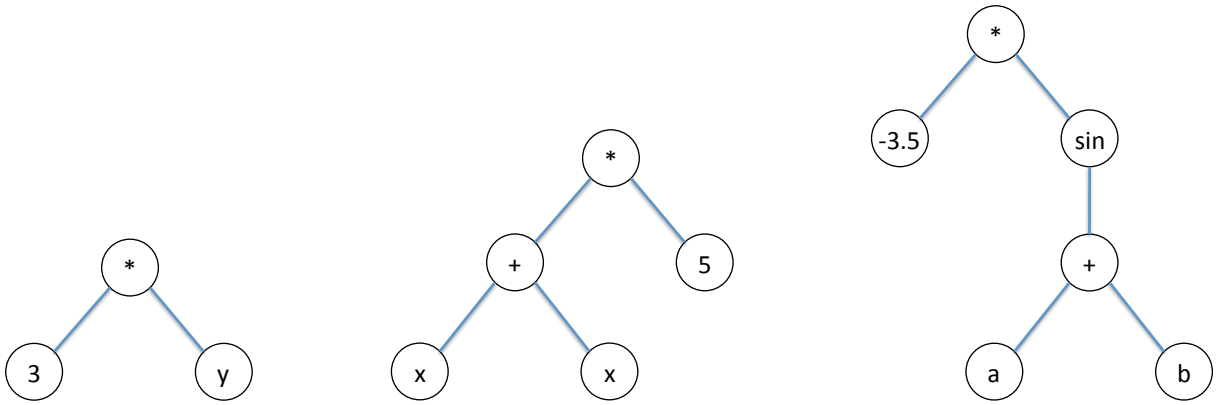

FIGURE 1 – Les arbres d'objets représentant E(x), F(x) et G(a,b)

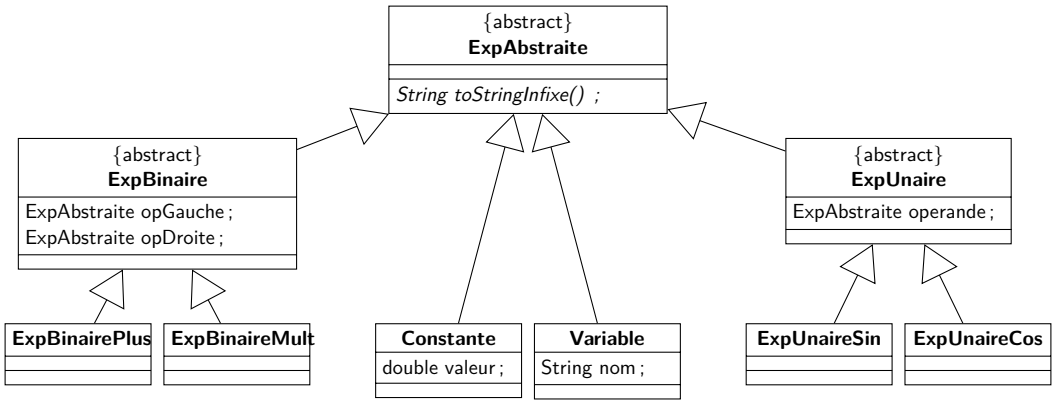

FIGURE 2 – Hiérarchie de classes pour les expressions

2 Classes des noeuds d'un arbre d'expression

Pour représenter les expressions, on implantera les classes Java décrites à la figure 2.

Dans ce diagramme, repérez les classes concrètes correspondant à chacun des opérateurs unaires ou binaires, et les classes permettant de représenter une variable de nom quelconque ("x", "var" ou "a" par exemple), ou une valeur constante.

Notez également que la classe abstraite `ExpBinaire` déclare deux arguments, pour les opérandes droit et gauche — qui sont des instances de `ExpAbstraite`. Le même raisonnement s'applique à la classe `ExpUnaire`, avec un seul opérande.

Ces classes permettent donc de représenter une expression quelconque par un arbre d'objets *Java. Dans cet arbre d'objets, les noeuds sont des instances de l'une des classes concrètes d'opérateurs, et les feuilles des instances des classes* `Constante` *ou* `Variable`*.*

Enfin, remarquez que la classe `ExpAbstraite` déclare une méthode abstraite : `String toStringInfixe()` dont le rôle est de retourner une représentation textuelle de l'expression, en notation infixée (usuelle) totalement parenthésée. Bien entendu, cette méthode abstraite doit être définie dans les sous-classes. L'exécution de cette méthode sur l'objet racine de l'arbre sera réalisée par un parcours infixé en profondeur de l'arbre des objets, récursif. Par exemple, dans la classe `ExpUnaireSin`, le pseudo-code de la méthode `String toStringInfixe()` est :

```
String s;
s += "sin(" ;
s += operande.toStringInfixe() ;
s+= ")";
retourner s
```

Pour les opérateurs binaires, et pour simplifier, on utilisera systématiquement des parenthèses. Par exemple, la chaîne représentant l'expression `F` sera :

$$((x + x) * 5) \tag{8}$$

Question 1 Implantez chacune des classes du diagramme de la figure 2. Prenez garde à la visibilité des attributs. Définissez le constructeur (un seul suffit dans chaque classe !) et les accesseurs adéquats. Pour choisir les paramètres du constructeur de chaque classe, regardez le premier programme de test de la question suivante. Il n'y a pas besoin de définir de modifieur (`set`) dans ces classes.

Implantez la méthode `String toStringInfixe()` dans toutes les classes concrètes de la hiérarchie.

Question 2 Redéfinissez la méthode `String toString()` dans la classe `ExpAbstraite` (et uniquement dans cette classe !) de telle sorte qu'elle retourne une chaine conforme à `"Je suis une expression et me voila en notation infixée : <ici, la notation infixée>"`.

Question 3 Testez vos classes au moyen du programme de test suivant (fichier `.java` disponible sur *Chamilo*).

```
public class TestAffichageInfixe {
    public static void main(String[] args) {
        ExpAbstraite exp;

        // teste l'expression préfixée *  y  3
        exp = new ExpBinaireMult(new Variable("y"), new Constante(3)) ;
        System.out.println(exp.toStringInfixe());

```

```
9
10          // teste l'expression préfixée * + x x 5
11          exp = new ExpBinaireMult(
12                          new ExpBinairePlus(
13                                          new Variable("x"),
14                                          new Variable("x")
15                          ),
16                          new Constante(5)
17          );
18          System.out.println(exp.toStringInfixe());
19
20          // teste l'expression préfixée * -3.5 sin + a b
21          exp = new ExpBinaireMult(
22                      new Constante(-3.5),
23                      new ExpUnaireSin(
24                              new ExpBinairePlus(
25                                      new Variable("a"),
26                                      new Variable("b")
27                              )
28                      )
29          );
30          System.out.println(exp.toString());
31      }
32  }
```

3 Encore plus de factorisation ?

Il existe encore de la redondance dans les classes concrètes d'expressions (que vous verriez encore plus en ajoutant d'autres opérateurs : tan, carré, etc.). Par exemple pour l'affichage d'une expression binaire : tout est dans les filles, alors que le shéma *afficher l'opérande gauche, puis l'opérateur, puis l'opérande droit* est toujours identique. Il en va de même pour l'évaluation des expressions.

Question 4 Comment, en ajoutant des quelques méthodes, factoriser le plus de code possible au niveau des classes abstraites intermédiaires `ExpBinaire` et `ExpUnaire`, et ne garder dans les filles que ce qui leur est réellement spécifique ? Y a-t-il un intérêt par rapport à la visibilité des attributs ?

4 Evaluation d'une expression

Question 5 Association variable - valeur

Pour calculer la valeur une expression, il faut substituer chaque variable par une valeur réelle. Il faut donc pouvoir associer une valeur à chaque variable, sachant qu'il peut y avoir plusieurs variables.

Pour cela, on va d'abord écrire une classe d'environnement `Env` dont le rôle est de stocker les associations entre les noms de variables et les valeurs de chaque variable. Le diagramme de cette classe est donné dans la figure 3.

Implantez la classe `Env` - en prenant garde bien sûr à l'encapsulation.

Note 1 : le diagramme ne précise pas le/les attribut(s) de la classe. A vous de les définir !

Le plus simple est d'avoir recours à un attribut de type "table associative", par exemple un attribut `HashMap<String, Double>`. On rappelle que les tables associatives sont présentées

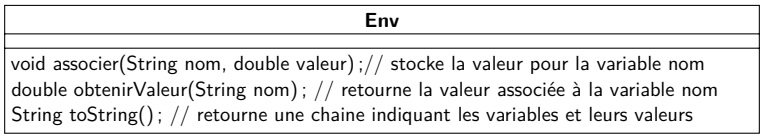

FIGURE 3 – Classe d'environnement pour stocker les associations entre nom de variable et valeur

dans le document "Introduction aux Collections" du TP temps libre, disponible sur *Chamilo*. Consultez également la Javadoc de `HashMap` : http://docs.oracle.com/javase/7/docs/api/java/util/HashMap.html

Si la notion de Collection Java est trop obscure pour vous pour le moment, vous pouvez aussi utiliser deux tableaux, même si cela demande plus de travail et n'est pas optimal : le premier tableau contiendra les clés (tableau de `String`) et le second les valeurs dans le même ordre (tableau de `double`). Pour la méthode `double obtenirValeur(String nom)` il faut alors parcourir le premier tableau pour trouver l'indice de la clé, et retourner la valeur stockée dans le second tableau au même indice.

Note 2 : notez que la méthode `obtenirValeur()` ne peut pas faire son travail si la variable n'est pas encore connue dans l'environnement. Lever une exception serait donc dans ce cas approprié...

Question 6 Méthode d'évaluation

Déclarez dans la classe `ExpAbstraite` une nouvelle méthode abstraite `double evaluer(Env env)`, dont le rôle est de calculer la valeur de l'expression en fonction des valeurs des variables stockées dans l'environnement `env` passé en paramètre.

Implantez la méthode `double evaluer(Env env)` dans chacune des classes concrètes de la hiérarchie.

Note : bien sûr, il n'est pas possible d'évaluer une expression si l'une des variables utilisée par l'expression n'existe pas dans l'environnement. Dans ce cas, lever une exception ou propager l'exception levée par la méthode `obtenirValeur` de l'environnement paraitrait approprié...

Question 7 Testez la méthode `evaluer` au moyen de la classe `TestEval.java` (disponible sur *Chamilo*). C'est le même programme que précédemment avec en plus l'évaluation des expressions, par exemple :

```
1 System.out.print("Valeur de l'expression, compte tenu des valeurs des
    variables :");
2 System.out.println( exp.evaluer(env) );
```

Note : sur *Chamilo*, vous trouverez aussi dans le fichier `TestParser.java` un autre programme de test un peu plus évolué, qui parse des expressions saisie par l'utilisateur au clavier. *A n'utiliser que si vous en avez le temps !!*

5 Extension possible : fonction dérivée

En supposant qu'on ne s'intéresse qu'aux expressions d'une seule variable, `x` par exemple, ajouter à la hiérarchie une méthode qui calcule la dérivée d'une expression donnée. Cette méthode `ExpAbstraite calculerDerivee()` retourne une nouvelle expression fonction de `x` correspondant à la fonction dérivée.

Variante : passer à cette méthode le nom de la variable par rapport à laquelle on veut calculer la dérivée.