Mise en pratique du visiteur

Objectifs

— modéliser des expressions arithmétiques et quelques traitements possibles,

— comprendre et mettre en œuvre le patron de conception *Visiteur*,

— mettre en évidence ses apports et ses limitations.

Exercice 1 : Compréhension des classes fournies

Dans un premier temps, utilisons et comprenons les classes fournies.

1.1. *La classe* `ExemplesAffichage`*.* Compiler le fichier `ExemplesAffichage.java` et exécuter la classe `ExemplesAffichage`. Ce programme affiche deux expressions sous forme infixe complètement parenthésée.

1.2. *Les expressions arithmétiques.*

1.2.1. Lister les classes qui font partie des expressions arithmétiques et dessiner le diagramme de classe correspondant.

1.2.2. Expliquer à quoi correspondent les méthodes de l'interface `VisiteurExpression`.

1.2.3. Expliquer ce que signifie « `R` » présent dans la signature de la méthode `accepter`.

1.2.4. Indiquer l'utilité de la valeur de retour des méthodes de l'interface `VisiteurExpression`.

1.3. *La classe* `AfficheurInfixe`*.* Expliquer la classe AfficheurInfixe.

Exercice 2 : Affichage en notation polonaise inverse

Les parenthèses permettent d'imposer un ordre de calcul particulier pour les sous-expressions d'une expression arithmétique donnée. La structure de l'arbre abstrait ne contient pas les parenthèses car elle impose elle-même l'ordre de calcul : les sous-expressions sont évaluées avant l'expression elle-même. La notation polonaise inverse (ou notation postfixe) est une représentation linéaire (par opposition à la représentation arborescente) qui impose également un ordre de calcul et permet de se passer des parenthèses.

Voici quelques exemples :

— « `x` » représente : x ;

— « `2 x +` » représente : $2 + x$;

— « `2 x + 3 *` » représente : $(2 + x) \times 3$;

— « `7 2 x + *` » représente : $7 \times (2 + x)$;

Comme il y a ambiguïté entre le « $-$ » unaire et le « $-$ » binaire, on utilisera le symbole `~` pour le « $-$ » unaire. Ainsi $-y$ se notera « `y ~` ».

Réaliser un visiteur `AfficheurPostfixe` imprimant une expression en notation polonaise inverse. Pour le tester, compléter la classe `ExemplesAffichage` puis utiliser la classe de test JUnit fournie (`AfficheurPostfixeTest`).

Exercice 3 : Hauteur de l'arbre d'une expression
La hauteur d'un arbre est la plus grande distance entre la racine de l'arbre et chacune de ses feuilles. Ainsi, la hauteur d'un arbre correspondant à une constante ou un accès à une variable est 1. La hauteur de $2 + x$ est 2 et celle de $7 \times (2 + x)$ est 3.

Voici la définition de la hauteur en s'appuyant sur la grammaire des expressions :

— hauteur($\mathrm{constante}$) = 1

— hauteur(ident) = 1

— hauteur($-E$) = 1 + hauteur(E)

— hauteur($E_1 + E_2$) = 1 + max(hauteur(E_1), hauteur(E_2))

— hauteur($E_1 * E_2$) = 1 + max(hauteur(E_1), hauteur(E_2))

Réaliser (et tester) un visiteur `CalculHauteur` qui calcule la hauteur de l'arbre d'une expression. La classe `CalculHauteurTest` est une classe de test JUnit de ce visiteur qui pourra être complétée.

Exercice 4 : Ajout de la soustraction
Pour autoriser la soustraction, on ajoute la règle de production suivante :

$$E \quad \rightarrow \quad E - E$$

4.1. Modifier le diagramme de classe pour introduire cette nouvelle forme d'expression.
4.2. Définir la (ou les) classe(s) supplémentaire(s) nécessaire(s).
4.3. Modifier les visiteurs déjà réalisés pour prendre en compte cette extension. On commencera par modifier l'interface et compiler l'application pour voir ce que signale le compilateur.

Exercice 5 : Valeur d'une expression
Réaliser (et tester) un visiteur `EvaluateurExpression` qui calcule la valeur d'une expression. On suppose que l'environnement, la valeur des variables, est stocké dans une structure de données associative de type `java.util.Map<String,Integer>`.

Si l'expression contient un accès à une variable non définie dans l'environnement, une exception sera levée.

La classe `EvaluateurExpressionTest` est une classe de test JUnit de ce visiteur.

Exercice 6 : Définition locale de variable
Les expressions actuelles ne permettent pas de définir de variables. Nous allons donc compléter leur grammaire. La nouvelle grammaire est donnée à la figure 1.
6.1. Modifier le diagramme de classe pour introduire cette nouvelle forme d'expression.
6.2. Définir la (ou les) classe(s) supplémentaire(s) nécessaire(s).
6.3. Modifier les visiteurs déjà réalisés pour prendre en compte cette extension. On commencera par modifier l'interface et compiler l'application pour voir ce que signale le compilateur.

Ainsi, « `5 x ni 2 x + tel` » est la notation polonaise inverse de « $\mathrm{let}\ x\ =\ 5\ \mathrm{in}\ 2\ +\ x$ ». On suppose que hauteur(let *ident* $= E_1\ \mathrm{in}\ E_2$) = 1 + max(hauteur(E_1), hauteur(E_2)).

Exercice 7 : Affichage infixe
La notation infixe est plus naturelle que la notation polonaise inverse. Cependant, il est nécessaire

$$
\begin{array}{rcl}
E & \rightarrow & \text{let } \textit{ident} = E \text{ in } E \\
 & | & E + E \\
 & | & E * E \\
 & | & -E \\
 & | & \text{constante} \\
 & | & \text{ident}
\end{array}
$$

FIGURE 1 – Grammaire des expressions pour gérer la définition de variables

d'utiliser des parenthèses. Dans un arbre abstrait, les parenthèses sont obligatoires si la priorité d'un nœud (en fait, celle de l'opérateur associé à ce nœud) est plus faible que la priorité du père de ce nœud.

Les affichages des exemples précédents donnent alors :

— x ;

— $2 + x$;

— $(2 + x) \times 3$;

— $7 \times (2 + x)$;

Réaliser (et tester) un visiteur `AfficheurInfixeMinimal` affichant une expression infixe en utilisant seulement les parenthèses nécessaires. La classe `AfficheurInfixeMinimalTest` est une classe de test JUnit.