Contrôle de l'arbre abstrait et production du code cible

Stéphane Malandain - Techniques de compilation - hepia

20 février 2017

Résumé

Ce document explique et guide le contrôle de l'arbre abstrait. Il représente donc l'analyse sémantique à proprement parler. Il donne ensuite quelques exemples et guide pour la production de code cible. Il reprend les chapitres 6 et 7 du livre *"Compilation des langages de programmation"* de Martine Gautier, aux éditions ellipses pour s'adapter au cadre défini par le langage HEPIAL.

Table des matières

Première partie

Le contrôle de l'arbre abstrait

Réalisé par un analyseur syntaxique, le contrôle de l'arbre abstrait s'appuie sur la table des symboles, utilisée essentiellement pour réaliser l'identification. Le résultat de celle-ci sert à typer les variables, donc les expressions, en fonction des règles sémantiques du langage. Parallèlement, les informations indispensables au générateur de code sont extraites de la table des symboles pour les attacher directement aux noeuds de l'arbre abstrait, ce qui allège la tâche du générateur de code. Si l'objectif principal est de relever les erreurs sémantiques, la décoration de l'arbre fait aussi partie intégrante de l'analyse sémantique.

1 Réalisation de l'analyse sémantique

Deux points de vues diffèrents peuvent guider la conception d'un analyseur sémantique :

— l'analyseur est assimilé à un ensemble de fonctions capables de contrôler respectivement une expression, une affectation, une itération, etc. ;
— l'analyseur est considéré comme un objet à part entière, à qui il faut s'adresser pour vérifier la sémantique de telle ou telle construction.

Dans la première solution, chaque classe représentant un noeud de l'arbre abstrait implante une méthode `controler` spécifique. Par exemple la méthode `controler` de la classe `affectation` contrôle la source et la destination et s'assure que les types de ces deux constructions sont conformes. Cette réalisation est une application du modèle de conception `Interpreteur`, qui suggère d'attacher à chaque classe du composite la méthode propre à son contrôle.

```
class Affectation extends Arithmetique {
    public void controler() {
        source().controler();
        Type tg = source().getType();
        destination().controler();
        Type td = destination().getType();
        if (tg.estConforme(td))
            type = tg;
        else
            ...
```

```
    } // controler
} // class Affectation
```

L'implémentation de la méthode `controler` dans la classe `Nombre` se contente de définir le type de la constante.

```
class Nombre extends Expression {
    public void controler() {
        type = TypeEntier.getInstance();
    } // controler
} // class Nombre
```

Le diagramme de séquence UML de la figure 1 illustre le déroulement simplifié des contrôles ou chaque rectangle symbolise un objet pour lequel est défini un comportement dans l'ordre chronologique, sous forme d'envois et de réception de messages, symbolisés par les flèches. Initié par le programme principal, ce contrôle est ensuite délégué (pour simpliifier) à une liste d'instructions, puis à une affectation, puis à une expression additive et à ses opérandes.

Dans la seconde solution, chaque classe implante la méthode `controler`,qui, cette fois, délègue la réalisation effective des contrôles à une instance d'une classe spécifique, `AnalyseurSemantique`,qui centralise à elle seule tous les contrôles.

```
class Affectation extends Arithmétique {
    public void controler() {
        AnalyseurSemantique.getInstance().controler(this);
    } // controler
} // class Affectation
```

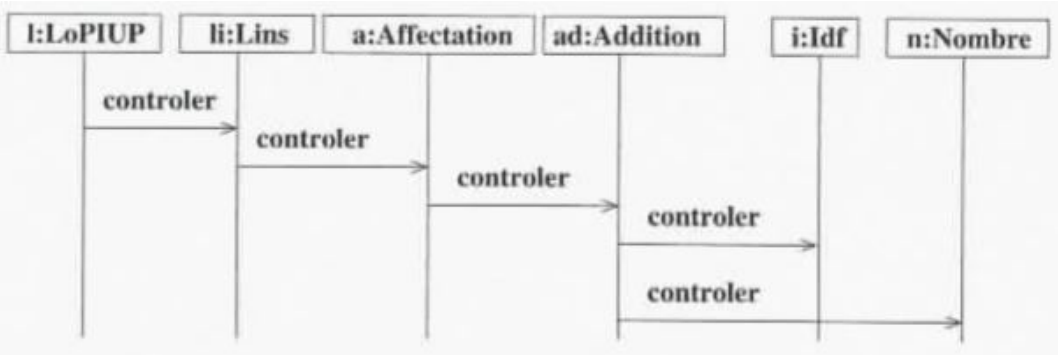

FIGURE 1 – Le contrôle est initié par le programme principal, puis délégué à chaque composant

```
class Nombre extends Expression {
    public void controler() {
        AnalyseurSemantique.getInstance().controler(this);
    } // controler
} // class Nombre

class AnalyseurSemantique {
    public void controler(Affectation a) {
        a.source().controler();
        Type tg = a.source().getType();
        a.destination().controler();
        Type td = a.destination().getType();
        if (tg.estConforme(td))
            a.setType(tg);
        else
            ...
    } // controler
} // class AnalyseurSemantique
```

Le diagramme de la figure 2, à comparer avec celui de la figure 1, illustre une partie des contrôles, liés à une affectation par exemple. Il met en évidence la

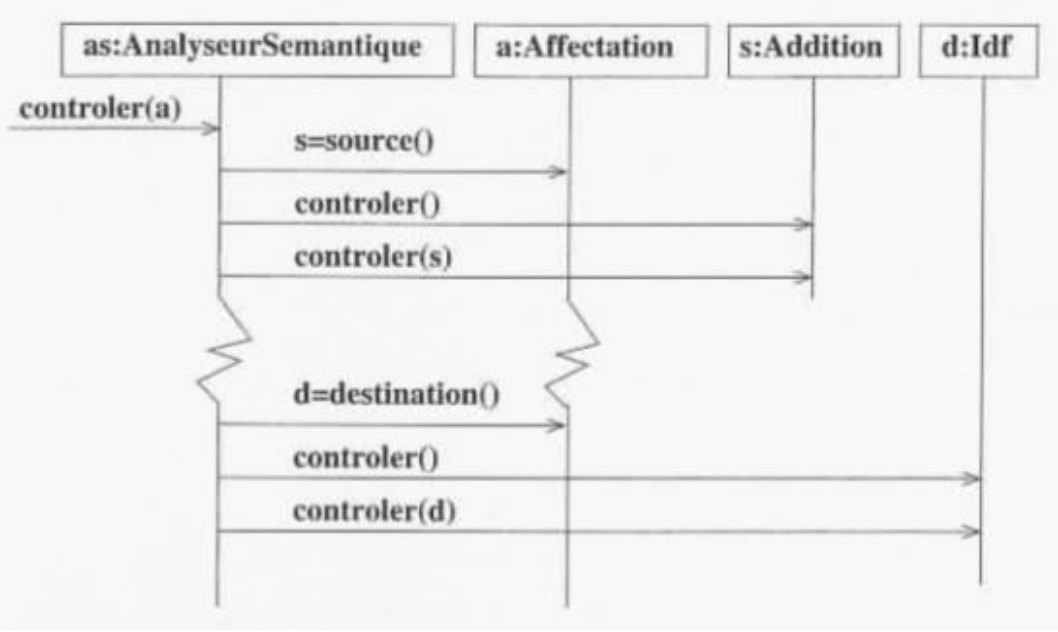

FIGURE 2 – Le contrôle est supervisé par la classe AnalyseurSemantique, qui puise les informations utiles dans les classes d'arbres abstrait

plus grande complexité de cette solution, qui rend la classe `AnalyseurSemantique` maître d'oeuvre des contrôles, à tous les niveaux. Cette dernière initie les contrôles et puise les informations utiles dans les noeuds de l'arbre abstrait.

Si plusieurs types de traitements doivent être attachés à chaque noeud, le même procédé est repris. Par exemple, on peut imaginer ajouter une opération `generer` prévue pour la production de code cible.

```
class Affectation extends Arithmetique {
    public void controler() {
        AnalyseurSemantique.getInstance().controler(this);
    } // controler
    public void generer() {
       GenerateurCible.getInstance().generer(this);
    } // generer
} // class Affectation
```

Cette solution alourdit le texte des classes d'arbres abstraits avec des méthodes similaires, n'apportant pas grand chose. L'application du modèle de conception `Visiteur` généralise la seconde solution proposée en donnant la possibilité d'ajouter à l'infini des traitements sur les instances d'une classe, sans avoir besoin de modifier explicitement la classe. Appliqué aux sous-classes de la classe `ArbreAbstrait`, il prévoit la définition d'une seule opération permettant d'accepter un visiteur, quel qu'il soit :

```
interface ArbreAbstrait  {
    Object accepter(Visiteur v) ;
} // interface ArbreAbstrait
```

La méthode `accepter` autorise la visite du visiteur `v` ; le modèle prévoit la possibilité de retour d'un résultat à la fin de la visite. Elle est définie de façon similaire dans toutes les classes d'arbre abstrait comme `Addition` et `Affectation`. (cf extrait de ces classes ci-dessous)

```
class Addition extends Arithmetique {
    public Object accepter(Visiteur v) {
        return v.visiter(this);
    } // accepter
} // class Addition

class Affectation extends Arithmetique {
    public Object accepter(Visiteur v) {
```

```
      return v.visiter(this);
   } // accepter
} // class Affectation
```

L'interface `Visiteur` propose les différentes méthodes de visite des noeuds de l'arbre abstrait :

```
interface Visiteur  {
    Object visiter( Addition a) ;
    Object visiter( Affectation a) ;
} // interface Visiteur
```

Pour chaque traitement à associer à un noeud de l'arbre abstrait, il reste à définir une implantation de `Visiteur` réalisant le traitement. Le visiteur `AnalyseurSemantique` est défini pour réaliser les contrôles sémantiques, comme l'illustre la figure 3 .

Quelle que soit la solution retenue, une fonction spécifique contrôle une construction donnée, la différence résidant simplement dans la classe qui contient cette fonction. Dans la première solution, chaque classe contient sa propre fonction de contrôle ; dans les deux autres solutions, les fonctions de contrôle sont centralisées dans une seule et même classe `AnalyseurSemantique`. Si le processus de développement du compilateur est incrémental, la première solution est attractive : une fois défini le noyau du langage décrivant complètement les expressions, les classes sont définitivement construites, sans avoir besoin d'y revenir lors de la construction des incréments suivants. A chaque incrément, on ne fait qu'ajouter de nouvelles classes d'arbres abstraits.

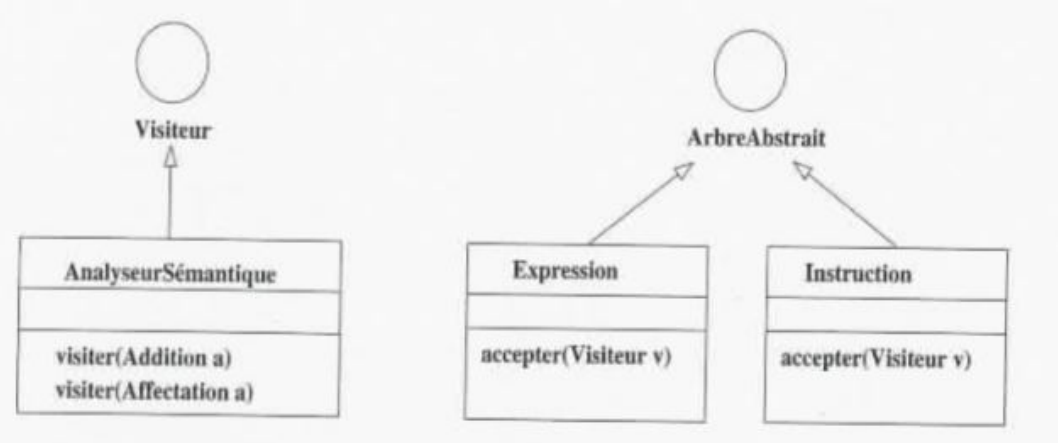

FIGURE 3 – Le visiteur AnalyseurSemantique contrôle chaque noeud de l'arbre abstrait

Dans l'autre solution, si les classes d'expressions sont aussi terminées après l'écriture du noyau initial, la classe `AnalyseurSemantique` doit être constamment complétée au cours de la compilation des noyaux suivants, pour intégrer le contrôle des nouvelles constructions. Le complément peut être réalisé par le biais de l'héritage pour conserver en l'état les classes développées pour le noyau initial. pour le k ième incrément développé, l'interface `visiteur` hérite de l'interface de l'incrément précédent et la complète avec le profil des méthodes de visite des noeuds de l'arbre traité dans l'incrément `k`. La classe `AnalyseurSemantique` hérite de la classe d'analyse sémantique de l'incrément précédent et implante l'interface `Visiteur` de l'incrément `k` (cf figure 4). Si le nombre d'incrément est important, cette solution peut s'avérer pesante, mais garantit une indépendance des noyaux.

Regrouper dans une seule et même classe les méthodes de contrôle assure une meilleure évolutivité si la définition du langage doit subir des changements ou si le développement d'un compilateur d'un autre langage doit être entrepris.
En outre, les créateurs du design pattern `Visiteur` suggèrent que ce dernier mérite d'être intégré dès que plusieurs visiteurs doivent être envisagés. Le contexte de développement d'un compilateur rentre très bien dans ce cadre, puisque la

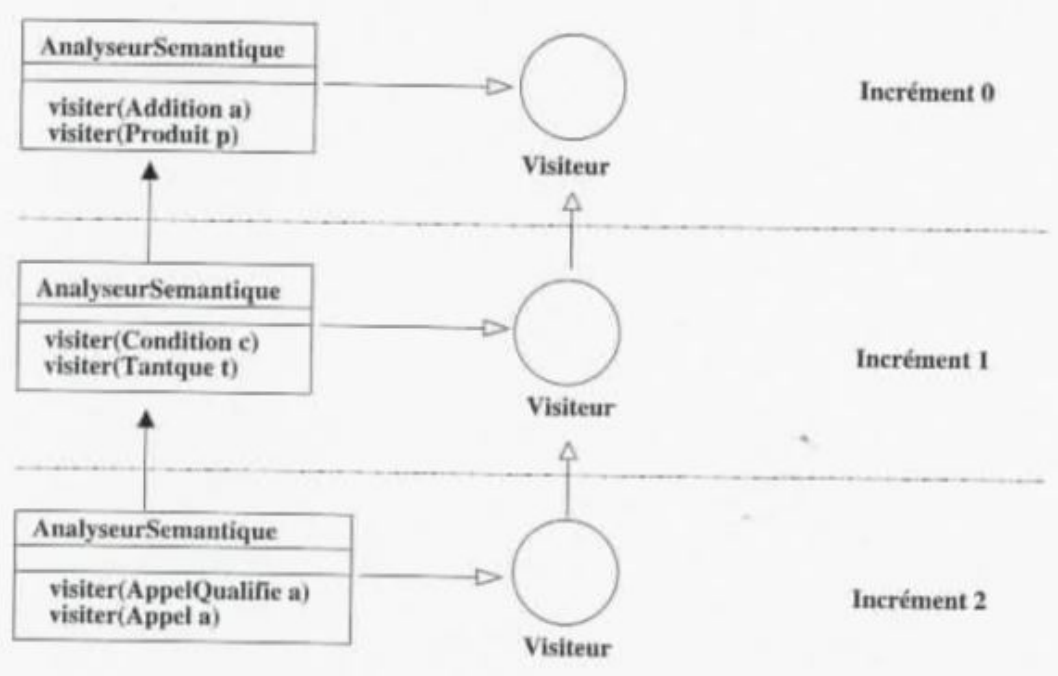

FIGURE 4 – Une classe AnalyseSemantique et une interface Visiteur sont définies pour chaque incrément, par héritage des versions antérieures.

génération de code peut donner lieu elle aussi à la définition d'un ou plusieurs visiteurs, selon la nature du code cible à produire. Il est même possible d'envisager un visiteur permettant de décompiler un arbre abstrait (c'est-à-dire retrouver une forme linéaire sous forme de texte) ou même de le visualiser.

Dans la suite, nous appliquons la seconde solution dans laquelle les fonctions de contrôle sont centralisées dans le visiteur `AnalyseurSemantique`.

1.1 Etat du compilateur

La majorité des contrôles à implanter dépendent essentiellement des composants de la construction étudiée. Par exemple, le contrôle d'un appel qualifié s'appuie sur les différents constituants de l'appel : le receveur, la fonction appelée et les paramètres effectifs. Dans certains cas, les contrôles sémantiques ne peuvent pas être réalisés localement, car le contexte d'utilisation de la construction est important.

Le singleton `EtatCompilateur` joue à nouveau un rôle dans l'implantation du contexte de compilation. Comme il sait déjà dans quelle phase se trouve le compilateur, il doit aussi savoir quelles sont les fonctions en cours d'analyse.

```
public class EtatCompilateur  {
// Consultation du singleton
public static EtatCompilateur getInstance() { .... }
// Consultation de l'état du compilateur
// compilateur en cours d'analyse syntaxique ?
public boolean analSyntEnCours() { ... }
// compilateur en cours d'analyse sémantique ?
public boolean analSemEnCours() { ... }
// Instructions retourne détectée ?
public boolean retourne() { ..... }
// première instruction analysée ?
public boolean premiereInstruction() { .... }
// compilateur en cours de génération de code ?
public boolean genererCode() { ... }
// function en cours de compilation ?
public SymboleFonction fonctionEnCours() { ... }
// Modification de l'etat du compilateur
// compilateur en cours d'analyse syntaxique
public void analSyntEnCours(boolean as) { ... }
// compilateur en cours d'analyse sémantique
```

```
public boolean analSemEnCours(boolean as) { ... }
// première instruction analysée
public void setPremiereInstruction(boolean Prem) { .... }
// compilateur en cours de génération de code
public void genererCodeEnCours(boolean gen) { ... }
// mémoriser la fonction en cours de compilation
public void setFonctionEnCours(SymboleFonction fct) { ... }
// Instructions retourne identifiée
public void setRetourne() { ..... }
} // class EtatCompilateur
```

Ces opérations vont être utilisées tout au long de l'analyse sémantique pour définir et identifier le contexte des contrôles.

1.2 Evaluation des expressions constantes

Eliminer les expressions constantes pour les remplacer par leur valeur simplifie le texte du code source et par conséquent les contrôles sémantiques et la production du code. Par exemple, avant de contrôler une affectation, il est plus efficace de chercher à calculer la valeur de la source si cela est possible ; de même, chercher à évaluer la condition d'une instruction conditionnelle peut conduire à simplifier l'instruction et à ne générer que le code des instructions de la partie *Alors* ou de la partie *Sinon*.

Ce calcul peut être réalisé par un nouveau visiteur d'arbre abstrait `Evaluateur` : ce singleton est envoyé visiter l'arbre pour tenter le calcul. Si celui-ci est possible, le résultat de la visite est la valeur de l'expression ; sinon l'évaluateur retourne la valeur `null`.

```
class Evaluateur implements Visiteur {
   public Object visiter (Nombre n) {
       return new Integer(n.valeur());
    } // visiter
    public Object visiter (Idf i) {
       return null;
    } // visiter
    public Object visiter (Addition a) {
        Object valG == a.gauche().accepter(this);
        if (valG == null) return null;
        Object valD == a.droit().accepter(this);
        if (valD == null) return null;
```

```
        int g = ((Integer)valG).intValue();
        int d = ((Integer)valD).intValue();
        return new Integer(g+d);
    } // visiter
    public Object visiter (Et e) {
        Object valG == e.gauche().accepter(this);
        if (valG != null) {
                int valintG = (( Integer )valG).intValue();
                return (valintG == 0)?valG:null;
        } // if
        Object valD == e.droit().accepter(this);
        if (valD != null) {
                int valintD = (( Integer )valD).intValue();
                return (valintD == 0)?valD:null;
        } // if
    } // visiter
} // class Evaluateur
```

La visite pour évaluation d'une instance de `Nombre` réussit toujours et retourne la valeur de la constante. la visite d'une instance de `Idf` échoue puisque qu'une variable n'a pas de valeur constante ; la méthode `visiter` dans ce cas retourne `null`.

La visite de tous les noeuds opérateurs arithmétiques et relationnels est réalisée de façon similaire : elle commence par la visite de l'opérande gauche. Si celui-ci n'est pas constant, l'expression ne peut pas être calculée et la visite s'arrête là. Ce principe est appliqué pour un noeud de type `Addition` : lorsque les deux opérandes sont constants, le résultat de la visite est la somme des deux opérandes.

Le cas des opérateurs booléens est différent puisque la valeur de l'un des deux opérandes peut suffire à calculer l'expression. La visite d'un noeud instance de la classe `Et` étudie la valeur de l'opérande gauche : s'il est constant et de valeur faux, l'expression entière prend la valeur faux quel que soit l'opérande droit. Si l'opérande gauche n'est pas constant, un calcul identique est opéré sur l'opérande droit. A noter que ce visiteur ne fait que calculer lorsque c'est possible, il ne cherche pas à simplifier l'expression.

1.3 Contrôle des constructions

1.3.1 Affectation

La sémantique de l'affectation du langage hepial impose que le type de la destination concorde avec celui de la source, comme l'illustre l'extrait suivant issue de la classe `AnalyseurSemantique`.

```
private static String ncf = "Source et dest. non conformes";
public Object visiter (Affectation a) {
       a.destination().accepter(this);
       Type typeDest = a.destination().type();
       Object v = a.source().accepter(Evaluateur.getInstance());
       if (v != null)
           a.setSource(new Nombre( (Integer)v, ligne() ) );
       a.source().accepter(this);
       Type typeSource = a.source().type();
       if  (! (typeSource.estConforme(typeDest)) )
           erreur(a, ncf);
       else
          a.setType(typeDest);
       return null;
} // visiter;
```

La visite commence par une visite de la partie gauche de l'affectation pour fixer son type. Ensuite, le singleton `Evaluateur` est envoyé visiter l'expression source pour tenter son calcul : si l'expression est constante, la valeur remplace l'expression elle-même. La visite de la source lui attribue ensuite un type. Le test de conformité entre le type de la destination et celui de la source est délégué à l'opération booléenne `estConforme` de la classe `Type`.

1.3.2 Instruction conditionnelle

La sémantique de l'instruction conditionnelle dans le langage hepial impose que le type de l'expression condition soit booléen.

```
private static String eb = "Expression booléenne attendue";
public Object visiter (Condition c) {
     Object v = c.condition().accepter(Evaluateur.getInstance());
     if (v != null)
        c.setValeurCondition( ((Integer) v).intValue() );
     else {
```

```
        c.condition().accepter(this);
        if (c.condition().type() != TypeBooleen.getInstance())
            erreur(c, eb);
      } // if
      c.alors().accepter(this);
      c.sinon().accepter(this);
      return null;
} // visiter;
```

Comme dans le cas de l'affectation, la visite de l'arbre abstrait d'une condition commence par un essai d'optimisation. Le singleton visiteur `Evaluateur` est envoyé visiter la condition pour en calculer la valeur. Si cela est possible, cette valeur remplace définitivement l'expression. Sinon la visite de la condition par le visiteur `AnalyseurSemantique` doit lui attribuer le type booléen. Pour finir, les deux listes d'instructions formant les parties *Alors* et *Sinon* sont visitées par l'analyseur sémantique.

1.4 Contrôle des expressions

Pour le langage hepial, le contrôle des expressions se résume à la vérification des règles de typage. Un arbre abstrait d'expression s'appuie sur le typage de ses opérandes. Pour implanter les différentes méthodes de contrôle, nous faisons l'hypothèse que chacune d'elles définit le type de l'arbre qu'elle visite, en positionnant un champ `type`, consultable par la méthode `getType`.

1.4.1 Feuilles de l'arbre abstrait

La visite de l'analyseur sémantique sur une feuille de type `Idf` se réduit à l'identification de la variable utilisée comme le montre l'extrait suivant issue de la classe `AnalyseurSemantique`.

```
private static String nd = "Identificateur non déclaré";
private static TDS tds = TDS.getInstance();
public Object visiter (Idf i) {
    // On cherche d'abord une variable locale ou un paramètre
    EntreeVarLocPar evlp = new EntreeVarLocPar(i);
    Symbole symb = tds.identifier(evlp);
    if (symb == null) {
          erreur (i, nd);
    i.setSymbole(symb);
    return null;
} // visiter;
```

Selon les règles sémantiques, l'identification d'une feuille `Idf` porte en priorité sur une variable locale ou un paramètre : cette recherche est déléguée à la méthode `identifier` de la classe `TDS`. En cas de recherche fructueuse, le symbole correspondant est attaché à l'identificateur, sinon le gestionnaire reçoit un message d'erreur.

La visite d'une feuille `Nombre` se contente de décorer l'instance avec son type. Celui-ci est défini par l'instance unique de `TypeEntier`.

```
public Object visiter (Nombre n) {
    n.setType( TypeEntier.getInstance());
    return null;
} // visiter;
```

1.4.2 Opérateurs arithmétiques

L'extrait suivant de la classe `AnalyseurSemantique` montre comment réaliser la visite des noeuds correspondant à des opérateurs arithmétiques.

```
private static String ti = "Typage incorrect";
public Object visiter (Arithmetique a) {
    a.gauche().accepter(this);
    Type tg = a.gauche().getType();
    a.droit().accepter(this);
    Type td = a.droit().getType();
    if (tg.estConforme(td))
        a.setType(tg);
    else
        erreur(a, ti);
     return null;
} // visiter;
```

Comme les règles sémantiques des quatre opérandes arithmétiques sont identiques, le contrôle est factorisé dans la méthode `visiter(Arithmetique a)`. Il se résume à la visite des deux opérandes, à la consultation de leur type calculé lors de la visite pour les comparer. La comparaison est déléguée à la méthode `estConforme` définie dans les sous-classes de `Type`. Si aucune erreur n'est détectée, le type des opérandes définit le type du noeud visité `a`. Le choix est fait ici de ne retourner aucun résultat, mais plutôt d'attacher une information au noeud visité pour décorer l'arbre.

Deuxième partie

De l'arbre abstrait au texte cible

Réalisée par un générateur de code, la production de code cible s'appuie sur l'arbre abstrait décoré lors de l'analyse sémantique, à partir des informations contenues dans la table des symboles. Si les étapes précédentes, analyses lexicale, syntaxique et sémantique se sont déroulées avec succès, toutes les informations utiles à la génération de code sont calculées et mémorisées dans l'arbre abstrait. Générer le code cible correspondant au texte source consiste à parcourir cet arbre et à produire le code spécifique à chaque construction utilisée.

1 Réalisation de la génération de code

La conception d'un générateur de code peut-être envisagée sous deux aspects différents, comme celle d'un analyseur sémantique :

— Le générateur de code est assimilé à un ensemble de fonctions capables de traduire chaque construction du langage source ;

— Le générateur de code est considéré comme un objet à qui il faut s'adresser pour générer le code de telle ou telle construction.

Dans le premier cas, chaque classe de l'arbre abstrait propose la méthode `controler` pour les contrôles sémantiques et la méthode `generer` pour la production de code. La classe `GenerateurByteCode` est une nouvelle implantation de l'interface `Visiteur`,destinée à visiter chaque noeud de l'arbre abstrait dans le but de produire le code correspondant.

Comme dans le cas de l'analyseur sémantique, cette seconde technique permet de centraliser les méthodes de production de code, ce qui favorise à moindre coûts la production de codes cibles différents. Dans un soucis de cohérence, nous adoptons la même solution que celle adoptée pour la réalisation de l'analyseur sémantique.

2 Construction du fichier cible

2.1 Structure d'un fichier cible écrit en bytecode

Un fichier cible écrit en bytecode contient la traduction du texte d'une seule classe, même si le fichier source en contient plusieurs. Cela permet de compiler chaque classe indépendamment mais en contrepartie impose que chaque fichier soit identifiable pour être retrouvé et utilisé lors de l'exécution. Par exemple, lors de la compilation de la classe C, le compilateur Java produit un fichier bytecode identifiant la classe et sa super classe :

```
.class public C
.super java/lang/Object
```

Après cette entête, on retrouve séquentielement le bytecode de chaque méthode, dans un ordre quelconque.

La production du code d'un système de classes se fait classe par classe, une classe par fichier comme l'extrait de la classe `GenerateurByteCode` suivant.

Dans le cas du langage hepial, bien sur, il faut adapter ce code en conséquence, puisque hepial n'est pas orienté objet.

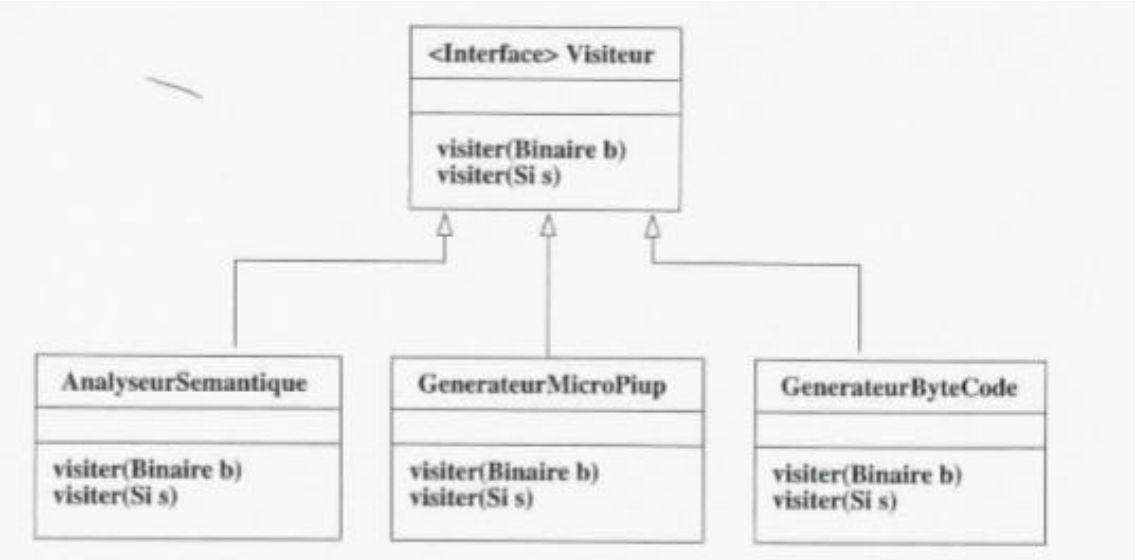

FIGURE 5 – les visiteurs AnalyseurSemantique, GenerateurMicroPIUP (inutile dans notre cas) et GenerateurByteCode implantent l'interface Visiteur

```
class GenerateurByteCode extends Visiteur {
   private static TexteCible cible = TexteCible.getInstance()
   public Object visiter(Systeme s) {
       // Code cible de chaque classe
       Enumeration enum = s.classes().elements();
       while (enum.hasMoreElements())
           ((Classe)enum.nextElement()).accepter(this);
       return null;
    } // visiter
    public Object visiter(Classe c) {
        cible.debut(c.idf());
        cible.add(".class public "+ c.idf().nom());
        Idf super = c.superClasse();
        if (super == null)
            cible.add(".super java/lang/Object");
        else
            cible.add(".super "+super.nom());
        // Code cible des constructeurs
        Enumeration enumc = c.constructeurs().elements();
        while (enumc.hasMoreElements())  {
            ((Constructeur)enumc.nextElement()).accepter(this);
        // Code cible des methodes
        Enumeration enumm = methodes.elements();
        while (enumm.hasMoreElements())  {
            ((Methode)enumm.nextElement()).accepter(this);
        cible.fin(c.idf());
        return null;
    } // visiter
} // class GenerateurByteCode
```

La production effective du code cible est réalisé par le singleton `TexteCible`, qui est en mesure de constituer le fichier cible, ligne par ligne. Dans un langage à objet, il faut le prévenir lors de la visite d'une classe qu il doit entamer la constitution d'un nouveau fichier cible. Ce n'est pas le cas avec hepial. Vient ensuite la production de l'entête, fonction de la présence ou de l'absence d'une super-classe déclarée, suivie de celle des constructeurs et des méthodes.

La génération du code d'une méthode peut se résumer à l'extrait de code suivant :

```
public Object visiter(Methode m) {
      SymboleMethode sm = m.symbole();
      cible.add(".method public "+sm.profil());
      cible.add(".limit locals "+sm.nbVarLoc());
      cible.add(" limit stack "+sm.taillePileOp());
      m.symbole().variablesLocales.accepter(this);
      m.symbole().instructions.accepter(this);
      cible.add(".end method");
      return null;
 } // visiter
```

Il suffit de produire l'entête de la méthode, les directives de déclarations des variables locales et de la taille de la pile des opérandes, suivent les déclarations des variables locales et le texte cible des instructions.

2.2 Génération du code de l'instanciation

voici un extrait pour la génération de bytecode concernant l'instanciation d'un objet.

```
public Object visiter(Nouveau n) {
      cible.add(" new "+n.classe();
      cible.add(" dup");
      // code cible qui empile les paramètres
      n.parametres().accepter(this);
      // code cible de l'appel du constructeur
      cible.add(" invokespecial "+n.classe()+"/$<$init$>$"+n.profil());
      return null;
 } // visiter
```

Les lignes 2 et 3 produisent le code qui crée une instance et empile son adresse dans la pile des opérandes. Suit la production du code qui empile les paramètres, réalisée par une visite de ceux-ci. La visite d'un noeud `Nouveau` se termine par la production de code de l'appel du constructeur.

2.3 Génération du code de l'affectation

La production de code cible suppose que les deux opérandes de l'affectation, source et destinations sont des entiers. Cela conditionne en effet l'utilisation des mnémoniques.

```
public Object visiter(Affectation a) {
     // code cible qui calcule la source
     a.source().accepter(this);
     // code cible de l'affectation
     int numRec = a.numeroReceveur();
     if (numRec <= 3)
         cible.add(" istore_ "+numRec);
     else
         cible.add(" istore "+numRec);
     return null;
 } // visiter
```

2.4 Génération de l'appel qualifié

Voici un autre exemple, toujours extrait de la classe `GenerateurByteCode` montrant comment générer un appel qualifié, dont vous pouvez vos inspirer pour l'appel de fonction.

```
public Object visiter(AppelQualifie a) {
     // code cible empile les parametres
     a.parametres().accepter(this);
     // code cible de l'appel avec liaison dynamique
     cible.add("  aload "+a.numeroReceveur());
     cible.add(" invokevirtual "+a.nomClasse()+"/"+a.idf());
     return null;
 } // visiter
```

La visite des paramètres produit le code qui les calcule et les empile. Vient ensuite la production de code qui empile le receveur et qui réalise explicitement la liaison dynamique.

2.5 Génération de code des expressions entières

L'étude du bytecode d'une expression entière est décrit dans le document explicitant la machine virtuelle java et le bytecode, disponible sur dokeos. L'opérande gauche d'un opérateur est calculé puis empilé, l'opérande droit est calculé et enfin l'opération proprement dite est réalisée. La pile des opérandes est supposée infinie ; elle est systématiquement utilisée.

Pour produire le code cible correspondant, il faut définir d'une part les opérations `visiter(Nombre n)` et `visiter(Idf i)` mais aussi les opérations correspondant à

chaque opérateur dans la classe `GenerateurByteCode`. Comme ces opérations sont similaires, au mnémonique près, une factorisation est souhaitable dans la méthode de visite d'un noeud `Binaire`. (cf extrait de la classe `GenerateurByteCode` suivant).

```
  public Object visiter(Nombre n) {
       int valeur = n.valeur();
       if (valeur <= 5)
           cible.add(" iconst_"+valeur);
       else
           cible.add(" bipush "+valeur);
       return null;
   } // visiter

   public Object visiter(Idf i) {
        if (i.symbole() instanceof SymboleVarLocPar) {
            int rang = i.rang();
            if (rang<=3)
                cible.add(" iload_"+rang);
            else
                cible.add(" iload "+rang);
        } else {
             cible.add(" aload_0");
             cible.add(" getfield "+i.nomComplet());
        } // if
        return null;
   } // visiter

   public Object visiter(Binaire b) {
        b.gauche().accepter(this);
        b.droit().accepter(this);
        cible.add(b.mnemonique());
        return null;
    } // visiter
   public Object visiter(Division d) {
        b.gauche().accepter(this);
        b.droit().accepter(this);
        genererErreur("0");
        cible.add(" idiv");
        return null;
    } // visiter
```

La visite d'un noeud `Nombre` se résume à empiler la valeur de la constante, en utilisant le mnémonique approprié, fonction de la valeur de la constante. La visite d'un noeud `Idf` distingue le cas d'une variable locale ou d'une variable paramètre, accessible par un simple numéro et la cas d'une variable d'instance qu'il faut retrouver à partir du receveur (lignes 17 et 18).

La factorisation de la génération de codes des opérateurs arithmétiques dans la méthode `visiter(Binaire b)` présente des limites : dans le cas d'un opérateur '/', le compilateur doit produire le code qui s'assure que le diviseur est non nul. Définir une méthode spécifique paraît donc indispensable. Il faut produire un test supplémentaire de l'opérande droit après son calcul : la méthode `genererErreur` produit le code d'appel à un sous-programme d'erreur, si la valeur en sommet de pile a la valeur donnée en paramètre.