Construction de l'arbre abstrait et de la table des symboles

Stéphane Malandain - Techniques de compilation - hepia

20 février 2017

Résumé

Ce document explique et guide l'implémentation d'une table des symboles et d'un arbre abstrait pour la réalisation du compilateur du langage hepial. Il reprend les chapitres 4 et 5 du livre *"Compilation des langages de programmation"* de Martine Gautier, aux éditions ellipses pour s'adapter au cadre défini par le langage HEPIAL et l'utilisation des analyseurs lexicaux et syntaxiques, `jflex` et `Cup`.

Table des matières

Première partie

Table des symboles (TDS)

La table des symboles est une structure de données complémentaire à l'arbre abstrait pour mémoriser les informations relatives aux déclarations. Elle est consultée pour chaque identification.

1 Le type abstrait TDS[Entree, Symbole]

L'accès à l'information se fait à partir d'une clé. Deux clés peuvent avoir la même valeur, mais à une clé ne peut être attaché qu'une seule valeur. Les opérations sur une table des symboles sont les suivantes ;

— Créer une table vide
— Ajouter une nouvelle clé
— Supprimer une clé existante
— Consulter / modifier des clés existantes

Pour les langages à structure de blocs, il faut intégrer l'imbrication des blocs. L'adjonction d'une entrée se fait toujours dans le bloc courant, mais l'identification d'une déclaration peut-être amenée à parcourir tous les blocs englobant le bloc courant.

1.1 Opérations

— `creer` crée une table vide.

— `ajouter(t,e,s)` complète la table t avec la nouvelle entrée e attachée au symbole s ; si cette entrée est déjà présente dans le bloc courant, la fonction retourne `doubleDeclaration.`

— L'opération `identifier(t,e)` retourne le symbole attaché à l'entrée e dans le bloc courant ou l'un des blocs englobants ; si cette entrée n'existe pas, la fonction retourne `indefini.`

— Les opérations `entreebloc(t)` et `sortiebloc(t)` signalent respectivement l'entrée et la sortie de bloc, de sorte que le mécanisme d'identification puisse s'adapter en conséquence.

2 Définition des types Entree et Symbole

Pour chaque construction, on utilise à la fois des informations mémorisées ou calculées par le compilateur. Par exemple, pour réaliser le contrôle sémantique d'un appel de fonction, le compilateur doit connaître (entre autre) le type de retour de la fonction. Pour générer le bytecode d'une affectation, il doit connaître le numéro de la variable.
Il est également indispensable de mémoriser l'emplacement de la ligne de chaque déclaration dans le texte source pour que le gestionnaire d'erreurs produise des messages d'erreurs syntaxiques et sémantiques pertinents. Toutes ces informations sont soit directement extraites du texte source, soit calculées. Elles doivent être rangées dans la table des symboles puisque qu'elles sont principalement attachées aux déclarations. Si le type générique `TDS` peut être utilisé pour créer, consulter et modifier n'importe quelle table des symboles, dans n'importe quel langage, il n'en est pas de même pour les types `ENTREE` et `SYMBOLE`, dont il n'existe pas de définition unique, communes à tous les langages sources et cibles. Chaque langage doit donner lieu à de nouvelles définitions de ces deux types.
Nous nous plaçons donc dans le cas d'un langage impératif (hepial) à structure de blocs, autorisant la déclaration de variables de types entier ou booléen. Un seul espace des noms est ouvert (pas de surcharge). Un bloc est défini par une fonction. Le passage des paramètres se fait par valeur. Le code cible doit être écrit en bytecode java.
Comme il n'existe qu'un seul espace des noms, un identificateur sert à déterminer de façon unique une déclaration dans un bloc donné. Une entrée dans la table des symboles est un simple identificateur :

```
ENTREE = IDENT
```

Un symbole contient toutes les informations attachées à une déclaration, destinées au contrôle sémantique mais aussi à la génération du code. Dans tous les cas il inclut le numéro de ligne de la déclaration dans le texte source. Le tableau suivant récapitule les différentes sortes de déclarations.

Déclarations	Analyse sémantique	Génération de code
Variable locale	type (entier, booléen)	rang de déclaration
Variable paramètre	type (entier, booléen)	rang de décla. dans liste des param
Fonction	type du résultat (entier, booléen)	taille pile des opérandes
	type des param. (entier, booléen)	nombre de param. et var. locales

3 Table des symboles pour un langage impératif

3.1 Interface de la classe TDS

Nous nous intéressons à la construction de la table des symboles d'un programme source écrit dans un langage impératif à structure de blocs, nommé hepial. Cette étude commence donc par la définition des classes `TDS, Entree` et `Symbole,` utilisées ensuite dans les actions insérées dans la grammaire pour construire la table des symboles.
Définir l'interface d'une classe `Tds` à partir de la définition du type abstrait `TDS` ne pose guère de problème. Cette classe manipule des clés et des valeurs sous forme d'`Object` si nous n'utilisons pas la généricité. (Avec, il devient possible de préciser explicitement le type des entrées et des symboles dans la construction de la classe).
Il existe une seule instance de cette classe, construite et consultée tout au long du processus de compilation. Cette situation est résolue par le design pattern `Singleton` : l'instanciation de la classe est interdite à tous les clients ; la méthode de classe `getInstance` permet de consulter l'instance unique.

```
class TDS {
    // consultation du singleton
    public static TDS getInstance();
    // ajouter un couple entrée, symbole
    public void ajouter ( Entree e, Symbole s) ;
    // identifier une entrée dans un bloc ouvert
    public Symbole identifier(Entree e) ;
    // entrer dans un nouveau bloc
    public void entreeBloc() ;
    // sortir du bloc courant
    public void sortieBloc() ;
} // class TDS
```

- La méthode `identifier(e)` retourne une instance de `Symbole` si l'identification de l'entrée e est possible dans l'un des blocs englobants, retourne `null` sinon.
- les méthodes `entreeBloc` et `sortieBloc` n'ont aucun paramètre : la première ouvre un nouveau bloc qui devient le bloc courant ; la seconde ferme le bloc courant pour se replacer dans son bloc englobant.
- La méthode `ajouter` n'a pas de résultat ; si aucune double déclaration n'est détectée, la déclaration est attachée au bloc courant. En cas de double déclaration, l'erreur est signalée à un gestionnaire d'erreurs, capable de mémo-riser l'erreur et de la signaler en fin de compilation. Ce gestionnaire

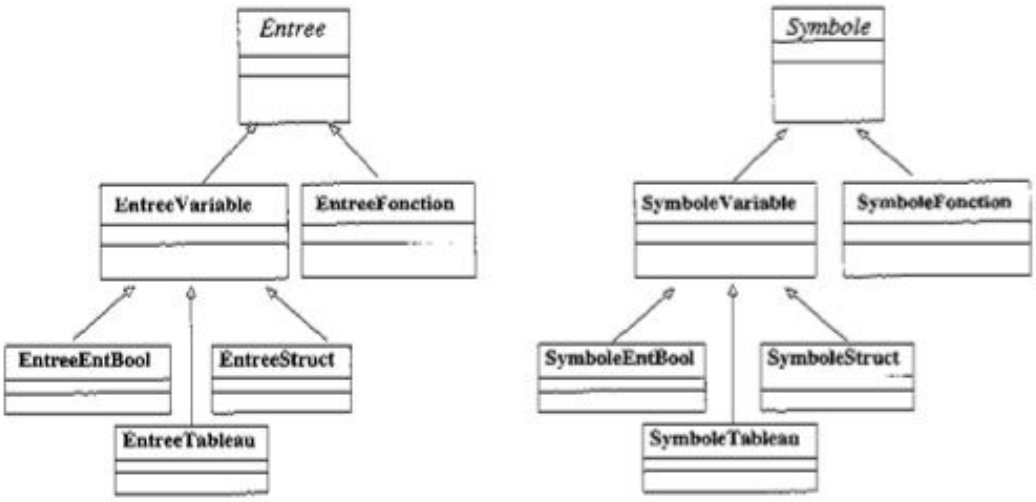

FIGURE 1 – les sous classes de Entree et Symbole

(encore un `Singleton`) s'approprie tous les problèmes inhérents à la gestion des erreurs sémantiques ; cela permet, entre autre, de poursuivre la détection de nouvelles erreurs sans arrêter la compilation dès la première erreur rencontrée.

3.2 Les classes Entree et Symbole

Il parait raisonnable de définir une classe abstraite `Entree` ; chaque entrée spécifique, comme les entrées de variable ou de fonction, donne lieu à la création d'une sous-classe. De la même façon, les différentes sortes de symboles donnent lieu à la création de sous-classes de la classe abstraite `Symbole`.

La classe `Entree` factorise les informations communes à toutes les entrées.

```
public abstract class Entree{
    // construire une entrée
    protected Entree(Ident ident) ;
    // Identificateur attaché à l'entrée
    public Ident ident() ;
} // class Entree
```

Les sous-classes de `Entree` n'offrent aucune fonctionnalité supplémentaire puisque dans chaque cas, l'identificateur de la déclaration suffit à identifier sans ambiguïté la déclaration. La classe `Symbole` factorise elle aussi les informations communes à tous les types de symboles, comme le numéro de ligne de la déclaration dans le texte source.

```
public abstract class Symbole {
    // construire un Symbole
    protected Symbole(int ligne) ;
    // Ligne dans le texte
    public int ligne() ;
} // class Symbole
```

Pour l'exemple, l'implantation de la sous-classe `SymboleTableau` est partiellement décrite ci-dessous :

```
public class SymboleTableau extends Symbole{
    // construire un symbole de tableau
    protected SymboleTableau(int ligne, Type t) { ... } ;
    // Description du type
    public Type type () { ... } ;
    // redéfinition de toString
    public String toString () { ... }
} // class SymboleTableau
```

Les autres sous-classes sont définies de façon similaire. Pour l'instant, elles ne proposent que des opérations permettant de définir et de modifier les champs. D'autres opérations pourront être ajoutées par la suite, soit pour satisfaire à une implantation spécifique de la table des symboles, soit pour répondre à des besoins de l'analyseur sémantique.

3.3 Construction de la table au cours d'une analyse descendante

Nous nous plaçons ici dans le cas d'une analyse syntaxique descendante, différente de l'analyse syntaxique ascendante mise en oeuvre par CUP. Cependant, cela donne une bonne idée sur la manière de procéder pour la construction de la table des symboles dans notre cas.

Celle-ci se fait à partir d'une table vide, construite par le constructeur TDS. Chaque nouvelle déclaration est ajoutée successivement en utilisant l'opération `ajouter`, sous réserve que les blocs soient ouverts et fermés à bon escient en utilisant les opérations `entreeBloc` et `sortieBloc`.
Soit l'extrait de grammaire suivant (avec le terminal 'NULL' représentant le mot vide `epsilon`), mettant en évidence uniquement les notions de blocs et de déclarations de variables de type simple, ainsi que les déclarations de fonctions. Les règles définissant les instructions et les expressions sont volontairement omises, puisque

ces constructions n'engendrent aucune adjonction dans la table des symboles. Les actions ajoutés dans les alternatives permettent à un analyseur syntaxique descendant de construire la table des symboles et de signaler la présence des doubles déclarations au gestionnaire d'erreurs.
Nous proposons ensuite un exemple de construction de la table dans le cas d'une analyse ascendante, mais avec le langage orienté objet `loPIUP` . Cet exemple a pour but également d'illustrer comment placer des actions n'importe ou dans la partie droite d'une règle, ce qui est possible avec un analyseur comme CUP.

```
BLOC    ->   debut (1) LDECV LINS (2) fin
LDECV   ->   DECL  LDECV
           | NULL
DECL    ->   TYPE ident (3) ';'
           | TYPE (4) ident ( PARAM ) debut LDECV  LINS (6) fin
PARAM   ->   TYPE (4) ident (7) SPARAM
           | NULL
SPARAM  ->   ',' TYPE (4) ident (7) SPARAM
           | NULL
TYPE    ->   booleen (8)
           | entier (9)
```

Avec les actions sémantiques suivantes :

```
(1) Tds.getInstance().entreeBloc();
(2) Tds.getInstance().sortieBloc();
(3) Entree e = new EntreeEntBool ( new Ident(uniteCourante));
    Symbole s = new SymboleEntBool(ligne, lastType);
    Tds.getInstance().ajouter(e,s);
(4) Type t = lastType;
(5) Ident ifonc = new Ident(uniteCourante);
    Tds.getInstance().entreeBloc()
    lastParam = new Parametres(spf);
(6) Tds.getInstance().sortieBloc();
    Entree em = new EntreeMethode(ifonc, lastParam);
    Symbole sm = new SymboleMethode(ligne, t, lastParam);
    Tds.getInstance().ajouter(em,sm);
(7) Ident ipf = new Ident(uniteCourante);
    Entree epf = new EntreeVarLocPar(ipf);
    Symbole spf = new SymboleVarPar(ligne, t);
```

```
    lastParam.ajouter(spf);
    Tds.getInstance().ajouter(epf,spf);
(8) lastType = TypeBooleen.getInstance();
(9) lastType = TypeEntier.getInstance();
```

L'ajout d'une entrée ne peut se faire que lorsque toutes les informations relatives au couple `<entree, symbole>` sont connues, c'est-à-dire à la fin de l'analyse d'une déclaration. Ainsi dans l'action (3) :

- l'identificateur de la variable (c'est-à-dire l'entrée) correspond à la dernière unité lexicale reconnue, notée ici `uniteCourante;` il sert de paramètre au constructeur de la classe `EntreeEntBool` ;
- le numéro de ligne de la déclaration est fixé par la variable `ligne` qui est une variable globale de l'analyseur syntaxique ;
- le type de la déclaration est fixé par les actions (8) et (9) ; il est mémorisé dans la variable globale de l'analyseur appelée `lastType.` Les types booleen et entier sont des singletons respectifs des classes `TypeBoolen` et `TypeEntier,` toutes deux sous-classes de la classe abstraite `Type`.

Pour la déclaration d'une fonction, plusieurs problèmes se combinent : il faut ajouter à la fois la déclaration de la fonction mais aussi celle des paramètres formels. L'entrée correspondant à la déclaration de la fonction est construite à partir des types des paramètres formels, donc elle n'est complètement définie qu'à la fin de l'analyse syntaxique de ceux-ci. Par ailleurs, si la fonction est déclarée dans le bloc courant, les paramètres formels, eux, sont déclarés dans le bloc d'imbrication supérieur. Si l'entrée de bloc est réalisée avant le début de l'analyse des paramètres (action (5)), ceux-ci peuvent être déclarés au fur et à mesure de leur analyse, dans le bloc adéquat. Il reste à attendre la sortie de bloc pour ajouter la déclaration de la fonction (action (6)).
L'action (7) a un objectif double : construire l'entrée et le symbole du paramètre pour pouvoir ajouter la déclaration du paramètre dans la table des symboles, et aussi mémoriser le symbole du paramètre formel analysé dans la liste `lastParam` de façon à construire le profil de la fonction. Cette liste, variable globale de l'analyseur, est utilisée à la fin de l'analyse de la fonction pour construire l'entrée de la fonction et ajouter la déclaration dans la table des symboles. A noter que cette adjonction tardive ne pose aucun problème puisque, lors de l'analyse syntaxique, seules les adjonctions sont prises en compte, l'identification étant du ressort de l'analyse sémantique. Nous invitons le lecteur souhaitant davantage de précisions sur le modèle par analyse syntaxique 'descendante' à consulter l'ouvrage de référence, cité en résumé de ce document.

3.4 Construction de la table au cours d'une analyse ascendante : extrait de source CUP

Nous proposons à titre d'exemple, un extrait du programme source CUP (figure 2, 3 et 4 permettant de construire la table des symboles du langage orienté objet, loPIUP. De nouveau nous invitons le lecteur souhaitant davantage de précisions sur le modèle pour un langage à objet à consulter l'ouvrage de référence cité en résumé de ce document.

3.5 Implantation de la classe TDS

Il est difficile de construire une classe `TDS` universelle. Notre étude suppose donc un compilateur réalisé en trois passes, imposant que la table soit persistante pour pouvoir être transmise à l'analyseur sémantique à la fin de l'analyse syntaxique.

Une simple table ne suffit pas, il faut intégrer la notion d'imbrication de blocs. Une déclaration est complètement définie par le triplet `<entree, symbole, bloc>`. L'opération `ajouter` ajoute dans le bloc courant, ouvert lors du dernier appel de `entreebloc` . L'opération `identifier` ne cherche pas une entrée seulement dans le dernier bloc ouvert, mais aussi dans tous les blocs englobant. La table doit donc gérer d'une façon ou d'une autre cette imbrication des blocs.

La table est construite et utilisée par deux analyseurs différents :

- L'analyseur syntaxique crée la table en utilisant le constructeur, les opérations de gestion des blocs (`EntreeBloc` , `SortieBloc`) et l'opération d'adjonction (`ajouter`).
- L'analyseur sémantique consulte la table en utilisant les opérations de gestions des blocs et l'opération `identifier`.

Il peut s'avérer utile de savoir dans quelle passe se trouve le compilateur, au moment de l'exécution des opérations de gestion des blocs. Plusieurs implantations de la table des symboles sont possibles. Nous proposons une implantation de table des symboles globale comme l'illustre la figure 5.

Nous gèrons un dictionnaire qui, a une entrée, associe une suite de couples `<symbole, bloc>`. Pour faciliter l'identification ultérieure, il est souhaitable de maintenir cette liste de couples triée dans l'ordre d'imbrication. Le reflet de l'imbrication des blocs ouverts, indispensable à la réalisation de l'identification, est donné par une pile de numéros de blocs. A tout moment cette pile contient les numéros de blocs imbriqués encore ouverts, donnant ainsi accès aux déclarations visibles depuis le bloc courant.

```
SYST     ::= CLASSES:lc
    {: RESULT = new Systeme(lc, lcleft) ;    :};

CLASSES ::= CLASSES:lc  CLASSE:c
    {: lc.add(c) ; RESULT = lc ;     :}|
            CLASSE:c
    {: Vector v = new Vector() ; v.add(c) ; RESULT = v ; :} ;

CLASSE   ::= CLASS  IDF:i
    {: EntreeClasse ic = new EntreeClasse(i) ;
       SymboleClasse sc = new SymboleClasse(ileft, null) ;
       Classe cla = new Classe(i, null, ileft) ;
       TDS.getInstance().ajouter(ic, sc) ;
       TDS.getInstance().entreeClasse(i) ;
       EtatCompilateur.getInstance().setClasseEnCours(cla) ;
       RESULT = cla ;        :}
            LDECL   FIN
    {: TDS.getInstance().sortieClasse() ;    :}|

            CLASS  IDF:i  HERITE  IDF:a
    {: EntreeClasse ic = new EntreeClasse(i) ;
       SymboleClasse sc = new SymboleClasse(ileft, a) ;
       Classe cla = new Classe(i, a, ileft) ;
       TDS.getInstance().ajouter(ic, sc) ;
       TDS.getInstance().entreeClasse(i) ;
       EtatCompilateur.getInstance().setClasseEnCours(cla) ;
       RESULT = cla ;        :}
            LDECL   FIN
    {: TDS.getInstance().sortieClasse() ;    :} ;

LDECL    ::= DECL  LDECL  |  ;

DECL     ::= STATUT:s  TYPE:t  IDF:i   LIDF:li PTVIRG
    {: // Le premier champ est ajouté dans la TDS
       EntreeChamp ec = new EntreeChamp(i) ;
       SymboleChamp sc = new SymboleChamp(ileft, s, t) ;
       TDS.getInstance().ajouter(ec, sc) ;
```

FIGURE 2 – Extrait TDS - CUP pour un langage à objet (1)

```
        // Les suivants
        Enumeration enum = li.elements() ;
        while (enum.hasMoreElements()) {
            i = (Idf)(enum.nextElement()) ;
            ec = new EntreeChamp(i) ;
            sc = new SymboleChamp(ileft, s, t) ;
            TDS.getInstance().ajouter(ec, sc) ;
        } // while    :}|

              STATUT:s  IDF:i
    {: TDS.getInstance().entreeBloc() ;      :}
              PARFOR:lp  DEB  LDECLVAR  LIINSTR:li FIN
    {: Classe cc =
              EtatCompilateur.getInstance().classeEnCours()
        EntreeConstructeur ec = new EntreeConstructeur(i,lp) ;
        SymboleConstructeur sc =
                new SymboleConstructeur(ileft, i, s, lp, li) ;
        TDS.getInstance().sortieBloc() ;
        TDS.getInstance().ajouter(ec, sc) ;
        cc.ajouterConstructeur(ec, sc) ;      :}|

              STATUT:s  TYPE:t  IDF:i
    {: TDS.getInstance().entreeBloc() ;      :}
              PARFOR:lp  DEB  LDECLVAR  LIINSTR:li FIN
    {: Classe cc =
              EtatCompilateur.getInstance().classeEnCours() ;
        EntreeMethode em = new EntreeMethode(i, lp) ;
        SymboleMethode sm =
                new SymboleMethode(ileft, i, s, lp, t, li) ;
        TDS.getInstance().sortieBloc() ;
        TDS.getInstance().ajouter(em, sm) ;
        cc.ajouterMethode(em, sm) ;  :} ;

LIDF    ::= VIRG  IDF:i  LIDF:li
    {: li.add(0, i); RESULT = li ;  :}|
    {: RESULT = new Vector() ;      :} ;

STATUT  ::= PUB
    {: RESULT = Statut.spublic ;    :}|
    PRIV
    {: RESULT = Statut.sprive ;     :} ;
TYPE    ::= ENT
    {: RESULT = TypeEntier.getInstance() ;  :}|
    IDF:idf
    {: RESULT = new TypeClasse(idf) ;       :} ;
PARFOR  ::= PAROUV  LPAR:lp  PARFER
    {:  RESULT = lp ;       :}|
            PAROUV   PARFER
    {:  RESULT = new ParametresFormels() ;  :} ;
```

FIGURE 3 – Extrait TDS - CUP suite (2)

```
LPAR     ::= PAR:s
    {: ParametresFormels pf = new ParametresFormels() ;
       pf.add(s) ; RESULT = pf ;     :}|
            PAR:s  VIRG  LPAR:pf
    {: pf.add(s) ; RESULT = pf ;     :} ;
PAR      ::=  TYPE:t  IDF:i
    {: EntreeVarLocPar e = new EntreeVarLocPar(i) ;
       SymboleVarPar s = new SymboleVarPar(ileft, null, t) ;
       TDS.getInstance().ajouter(e, s) ;
       RESULT = t ; :} ;
LDECLVAR::= | LDECLVAR  DECLVAR ;
```

FIGURE 4 – Extrait TDS - CUP pour un langage à objet (3)

le dictionnaire et la pile sont utilisés conjointement :

— lors de l'entrée dans un bloc, un nouveau numéro de bloc est empilé. C'est le numéro de bloc courant.
— chaque ajout de déclaration se fait en tête de la liste attachée à l'entrée, en utilisant le numéro de bloc courant pour constituer le couple `<symbole, bloc>`.
— lors de la sortie d'un bloc, le sommet est dépilé.
— la réalisation du mécanisme d'identification nécessite un parcours en parallèle de la pile des blocs ouverts, et de la liste de couples attachés à l'entrée, ces deux structures étant triées par numéros de blocs décroissants.

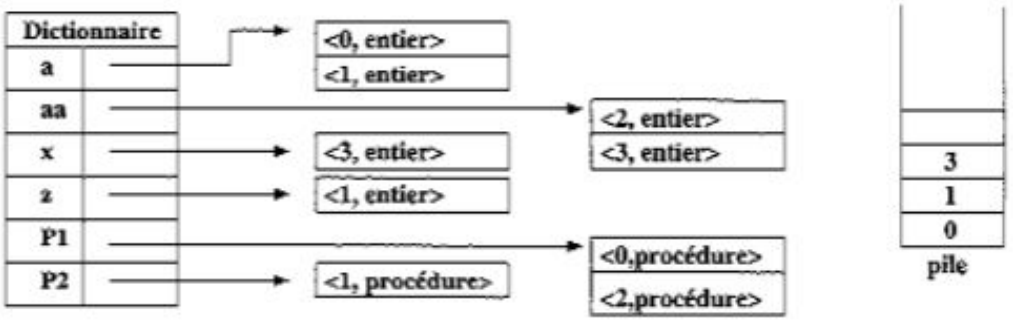

FIGURE 5 – Table des symboles globale

La classe `TDS` est implantée avec trois champs :

— le champ `pile` de type `Stack` est une pile d'entiers.
— le champ `dico` de type `HashMap` est un dictionnaire. L'entrée sert de clé. La valeur attachée à chaque clé est une pile de couples `<symbole, bloc>`.
— le champ `numeroBloc` de type `int` mémorise le numéro du bloc courant.

Les figures 6 et 7 donnent un extrait de la classe TDS.

Cette solution présente l'avantage de ne faire aucune distinction entre les modes de construction et de consultation de la table des symboles : entrer dans un bloc se résume toujours à empiler un nouveau numéro de bloc.

3.6 Implantation de la classe Type

Une instance de classe `Type` décrit n'importe quel type simple ou structuré. Définir une hiérarchie de sous-classes de `Type` pour décrire chaque type de données différent par une classe facilite l'extension de nouvelles descriptions de types de données. La figure 8 propose une hiérarchie de sous-classe de la classe `Type`,

Extrait de la classe TDS

```
import java.util.* ;
class TDS {
  // Champs privés
  private Stack pile ;
  private HashMap dico ;
  private int numeroBloc = -1 ;
  private static TDS instance = new TDS() ;
  // Consultation du singleton
  public static TDS getInstance(){
     return instance ;
  }// getInstance
  private TDS() {
    pile = new Stack() ;
    dico = new HashMap() ;
  }// TDS
  public void entreeBloc() {
    numeroBloc ++ ;
    pile.push(new Integer(numeroBloc)) ;
  }// entreeBloc
  public void sortieBloc() {
    pile.pop() ;
  } // sortieBloc
  public void ajouter(Entree e, Symbole s)  {
    Stack ls = (Stack)(dico.get(e)) ;
    if (ls == null) {
      // Nouvelle entrée, on crée une pile vide
```

FIGURE 6 – extrait de la classe TDS (1)

```
      ls = new Stack() ;
      ls.push(new Association(s, numeroBloc)) ;
      dico.put(e, ls) ;
    } else {
     Association premier = (Association)(ls.peek()) ;
     if (premier.numeroBloc() == numeroBloc) {
       // Double déclaration
       FlotErreurs.getInstance().add(s.ligne(),
                ErreurSemantique.doubleDecl(e, s.ligne())) ;
     } else
       // Adjonction dans une liste existante
        ls.push(new Association(s, numeroBloc)) ;
    } // if
  } // ajouter
  public Symbole identifier(Entree e) {
    Stack ls = (Stack)(dico.get(e)) ;
    if (ls == null) return null ;
    int indicePile = pile.size()-1 ;
    int indiceLs = ls.size()-1 ;
    boolean fin = false ;
    Object s = null ;
    int premListe, premPile ;
    Association assoc ;
    while (! fin && indicePile != -1 && indiceLs != -1 ) {
      assoc = (Association)(ls.get(indiceLs)) ;
      premListe = assoc.numeroBloc() ;
      premPile = ((Integer)(pile.get(indicePile))).intValue();
      if (premListe == premPile) {   // Symbole trouvé
        s = assoc.symbole() ;
        fin = true;
      } else if (premListe < premPile)
                // On avance dans la liste
                indiceLs-- ;
              else
                // On avance dans la pile de numéros de blocs
                indicePile-- ;
    } // while
    return s ;
  } // identifier
} // TDS
```

FIGURE 7 – extrait de la classe TDS (2)

décrivant pour ce qui nous concerne les types entier, booléen ou tableau. Il n'existe que des instances uniques des classes `TypeEntier` et `TypeBooleen`, alors qu'il existe autant d'instances de `TypeTableau` que de descriptions de types différents dans le texte source.

La classe abstraite `Type` propose simplement l'opération `estConforme` permettant de vérifier la conformité entre le type receveur et le type passé en paramètre ;

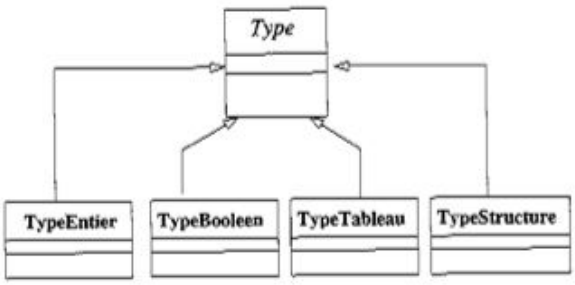

FIGURE 8 – les sous-classes de type

cette opération est principalement destinée à l'analyseur sémantique, pour réaliser les contrôles de types. Interface de la classe `Type` :

```
public abstract class Type {
    /** Conformité de 2 types
      *  @param other un autre type
      *  @return vrai si this est conforme à other
      */
    public abstract boolean estConforme( Type other );
} // class Type
```

L'opération `estConforme` est implantée de façon spécifique dans chaque sous-classe, en respect de la régle de conformité définie dans le langage à compiler. Par exemple, dans la classe `TypeEntier` (comme dans la classe `TypeBooleen`), l'implantation respecte la règle stricte qui impose l'égalité des deux types comparés. Les lignes suivantes donne un extrait de la définition de la classe `TypeEntier`.

```
public class TypeEntier extends Type {
     public boolean estConforme(Type other) {
            return other instanceof TypeEntier;
     } // estConforme
} // TypeEntier
```

Selon la règle d'équivalence structurelle ou nominale, l'implantation est plus ou moins délicate pour la classe `TypeTableau`. Dans le cas d'une équivalence structurelle, l'opération `estConforme` de la classe `TypeTableau` compare le type des éléments du tableau, le nombre de dimensions et les taille de chacune d'elles.

La figure 9 donne un extrait de l'implémentation de la classe `TypeTableau` qui permet de représenter des tableaux statiques pour lesquels sont mémorisés les différentes bornes inférieures et les bornes supérieures, les enjambées et la translation `Bik * ek`.

Extrait de la définition de la classe TypeTableau

```
class TypeTableau extends Type {
    protected Type type ;          // Type des éléments
    protected int bikek ;          // Somme des Bik * ek
    protected int[] borneInf ;     // Bornes inférieures
    protected int[] borneSup;      // Bornes supérieures
    protected int[] enjambees;     // Enjambées

    // Construire un symbole de tableau
    public TypeTableau(int ligne, Idf i, Type t, int bikek,
                       int[] binf, int[] bsup, int[] enj) {...}
    // Type des éléments
    public Type type() {...}
    // Nombre de dimensions
    public int nbDimensions() {...}
    // Somme des Bik * ek
    public int bikek() {...}
    // Borne inférieure de la kème dimension
    public int borneInferieure(int k) {...}
    // Borne supérieure de la kème dimension
    public int borneSuperieure(int k) {...}
    // Enjambée de la kème dimension
    public int enjambée(int k) {...}
    // Redéfinition de toString
    public String toString() {...}

    // Test de conformité
    public boolean estConforme(Type other) {
      if (!(other instanceof TypeTableau)) return false ;
      int nb, nbto ;
      TypeTableau to = (TypeTableau)other ;
      boolean res = (nbDimenions()==to.nbDimensions()) &&
                     type().estConforme(to.type()) ;
      for (int k=0; k<nbDimensions() && res ; k++) {
        nb = borneSuperieure(k)-borneInferieure(k) ;
        nbto = to.borneSuperieure(k)-to.borneInferieure(k) ;
        res = (nb == nbto) ;
      } // for
    } // estConforme
} // class TypeTableau
```

FIGURE 9 – Extrait de la définition de la classe TypeTableau

Deuxième partie

Arbre abstrait

1 Introduction

Un analyseur ascendant part des feuilles de l'arbre syntaxique, puis réduit plusieurs feuilles en un non-terminal, jusqu'à réduire une dernière fois et obtenir l'axiome. Le choix de la règle utilisée pour réduire n'est fait qu'une fois reconnus tous les éléments de la partie droite de la règle. Il est donc impossible - en théorie - de placer des actions ailleurs qu'en fin de règle.
Dans l'extrait de grammaire suivant

```
DECLAVAR ->    TYPE LIDENT ';'           (1)   /1/
             | TYPE ident '=' EXPR ';'   (2)   /2/
```

les actions ne seront éxécutées qu'à la fin de l'analyse complète de `DECLAVAR` : si l'alternative /1/ est choisie pour réduire, l'action (1) est exécutée, sinon, c'est l'action (2). Si ces actions doivent utiliser le résultat des actions exécutées lors de la réduction des règles décrivant `TYPE`, `LIDENT` ou `EXPR`, il est indispensable de mémoriser ces résultats dans une structure intermédiaire, gérée en pile. A chaque réduction, les résultats correspondant aux éléments de la partie droite sont dépilés pour être utilisés pour construire le résultat à empiler.

Si la théorie exige que les actions soient systématiquement placées en fin de règle, en pratique, le générateur d'analyseur ascendant CUP autorise l'insertion d'actions à des endroits quelconques. Cette possibilité nous permet de placer des actions n'importe ou dans la partie droite de la règle, simplifiant ainsi l'écriture des actions dans les grammaires complexes.

2 Construction de l'arbre abstrait

2.1 Les classes Expression et Instruction

Nous utilisons le design pattern `Composite` pour implémenter ce problème classique d'implantation d'arbre dans un langage à objets. Ce modèle de conception propose de définir trois classes :

— le composant : dans un arbre, tous les éléments sont des composants.
— le composite est un composant composé de plusieurs composants.

— la feuille est un composant sans fils

Appliqués à la représentation des arbres d'expressions, ce modèle de conception conduit à la définition d'une classe composant `Expression` ; de cette classe abstraite hérite autant de classes feuilles que d'opérandes possibles et autant de classes composites que d'opérateurs. Cette architecture permet de profiter des avantages de la liaison dynamique et de traiter un composite comme une feuille : l'application stricte de ce modèle architectural exige que l'interface des classes feuilles et composites soit commune.

la figure 9 propose une ébauche de ce graphe d'héritage mettant en application ce design pattern : les classes `Nombre` et `Ident` sont des feuilles ; les classes `Addition` et `Soustraction` sont des composites, puisque les instances de ces classes ont deux champs `gauche` et `droit` de type `Expression`.

Ce premier graphe incomplet n'intègre pas tous les opérateurs, mais toutes les classes correspondant à des opérateurs binaires vont être construites sur le même modèle, ce qui va entrainer une répétition des champs `gauche` et `droit` pour chacune d'elles. Il en sera de même pour les opérations permettant de construire et de modifier ces champs. Envisager une meilleure structuration du graphe est indispensable pour éviter cette duplication. Le graphe suivant (figure 10) tient compte de cette remarque.

Par exemple, la classe abstraire `Binaire` factorise le comportement des opérandes binaires, donc les champs `gauche` et `droit`, ainsi que les opérations destinées à consulter et modifier ces champs ; ces opérations implicites ne figurent pas dans le graphe d'héritage de façon à l'alléger. Les opérateurs peuvent encore être classés en deux catégories : les opérateurs arithmétiques (classe `Arithmetique`) et les opérateurs relationnels (classe `Relation`). Cette distinction permet de factoriser des comportements communs à l'une ou l'autre des deux catégories d'opérateurs. Le graphe d'héritage des classes d'instructions peut être construit selon le même modèle, comme l'illustre la figure 11.

La classe `Instruction`, comme la classe `Expression` est une implémentation particulière de la classe abstraite `ArbreAbstrait` :

— la classe abstraite `Instruction` est un composant.
— la classe `Bloc` est un composite.
— les classes `Affectation, Condition, Pour, Appel,` etc. sont des feuilles.

FIGURE 10 – graphe d'héritage complet des classes d'expressions

Toutes ces classes n'incluent pour l'instant que des opérations de construction et de consultation des champs. Elles pourront être complétées ultérieurement. Voici pour l'exemple, l'implantation du composite `Binaire` et de la feuille `Nombre`.

```
public abstract class Binaire extends Expression {
     protected Expression operandeGauche;   // opérande gauche
     protected Expression operandeDroit;        // opérande droit
     // Construire à partir des deux opérandes
     public Binaire (Expression g, Expression d, int lig)  {
           super(lig);
           operandeGauche = g;
           operandeDroit = d;
      } // binaire
      // consulter l'opérande gauche
      public Expression gauche() {
           return operandeGauche;
      } // gauche
      // consulter l'opérande droit
```

```
      public Expression droit() {
           return operandeDroit;
      } // droit
      // greffe un nouvel opérande gauche
      public void grefferGauche (Expression exp) {
           operandeGauche = exp;
      // grefferGauche
      // greffe un nouvel opérande droit
      public void grefferDroit (Expression exp) {
           operandeDroit = exp;
      // grefferDroit
      public String toString() {
           return "(" + operandeGauche+operateur()+operandeDroit+")";
      } // toString
      public abstract String operateur();
} // class Binaire


public class Nombre extends Expression {
     private int valeur;
     // Créer à partir de la valeur du nombre
     public Nombre (Integer val, int lig) {
          super(lig);
          this.valeur = val.intValue();
     } // Nombre
     // valeur de l'expression
     public int valeur() {
         return valeur;
     } // valeur
     public String toString() {
         return ""+valeur;
      } // toString
} // Nombre
```

2.2 Création de l'arbre au cours d'une analyse ascendante

2.2.1 Arbre abstrait d'une liste d'instructions

Par convention, nous décidons que l'arbre abstrait construit pour une alternative est toujours au sommet de la pile des arbres géré par l'analyseur.

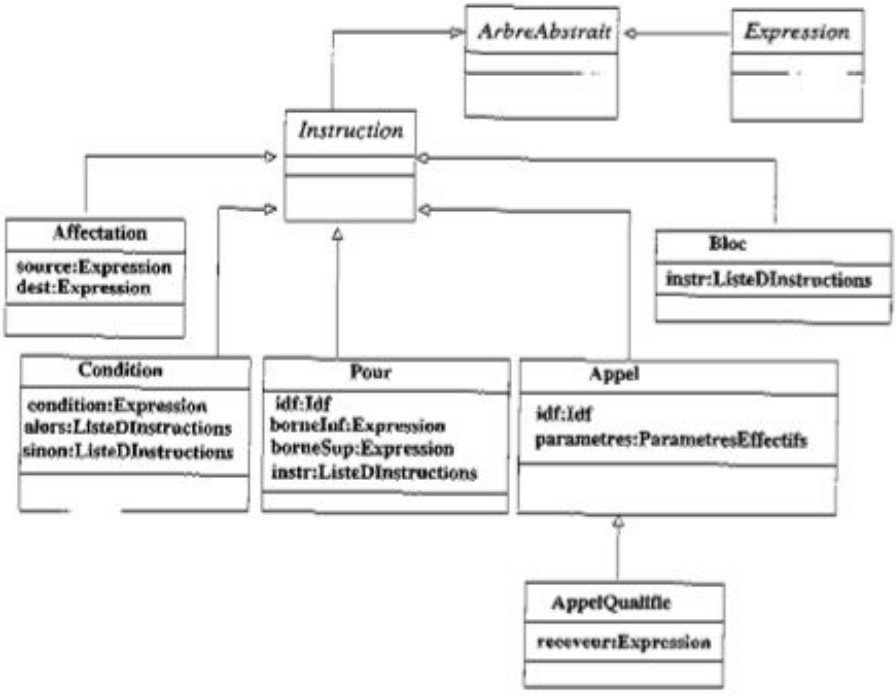

FIGURE 11 – Les classes Expression et Instruction sont des sous-classes de la classe abstraite ArbreAbstrait

Pour créer un arbre abstrait de liste d'instructions sous forme d'un arbre n-aire, il faut compléter les règles de description du non-terminal `LINSTR` [1] en ajoutant des actions uniquement à la fin des deux alternatives

```
LINSTR ->   LINSTR INSTR  (1)
        |   NULL  (2)
```

avec les actions définies par :

```
(1)   Instr i = (Instr) (pilesArbres.depiler());
      Linstr li = (Linstr) (pilesArbres.depiler());
      pileArbres.empiler(new Linstr(i, li));
 (2)  pileArbres.empiler(new Linstr());
```

1. LINSTR : := LINSTR INSTR correspond à INSTR* défini dans la grammaire et 'NULL' représentant le mot vide epsilon.

Une fois encore, ces actions font l'hypothèse que si l'analyse d'une instruction se passe bien (dérivations de INSTR), l'arbre abstrait de l'instruction est construit et accessible. L'arbre est rangé dans la pile des arbres, nommée ici `pileArbres` (alors que pour une analyse descendante, l'arbre est rangé dans une variable globale puis sauvegardé dans une variable locale).
L'analyse réussie de chaque non-terminal entraîne l'empilement de l'arbre abstrait correspondant. A la fin de chaque règle, la création de l'arbre abstrait à empiler utilise les différents constituants de l'arbre, eux-mêmes déjà empilés lors des analyses précédentes. A la fin de l'analyse, la pile ne contient plus qu'un seule arbre, celui du texte complet.

2.2.2 Arbre abstrait d'une affectation

Pour créer l'arbre abstrait d'une affectation, une seule action suffit à la fin de l'alternative décrivant `AFFECTATION`.

```
INSTR       ->  AFFECTATION
AFFECTATION ->  ACCES '=' EXPR ';'  (3)
```

avec les actions définies par

```
(3)   Expr source = (Expr) (pilesArbres.depiler());
      Ident dest = (Ident)(pilesArbres.depiler());
      pileArbres.empiler(new Affectation(dest, source));
```

L'action (3) s'appuie sur l'hypothèse que dans les alternatives décrivant `ACCES` et `EXPR`, l'arbre abstrait est correctement construit et empilé. Il suffit donc de dépiler l'arbre de l'expression rangé dans la variable `source`, de dépiler également l'arbre mémorisant la variable destination de l'affectation (rangé dans la variable `dest`). Le respect des hypothèses de travail nous conduit ensuite à créer et à empiler un arbre abstrait pour l'affectation.

2.2.3 Arbre abstrait d'une instructions conditionnelle

La construction de l'arbre abstrait d'une instruction conditionnelle suit le même principe que les exemples précédents. A la fin de chaque alternative décrivant les deux variantes possibles, une action récupère les composants de l'instruction conditionnelle pour les enraciner.

```
INSTR     ->  CONDITION
CONDITION ->  si EXPR alors LINSTR sinon LINSTR finsi  (4)
           |  si EXPR alors LINSTR finsi  (5)
```

avec les actions définies par

```
(4) Linstr sinon = (Linstr) (pilesArbres.depiler());
    Linstr alors = (Linstr) (pilesArbres.depiler());
    Expr ec = (Expr) (pilesArbres.depiler());
    pileArbres.empiler(new Si(ec, alors, sinon));
(5) Linstr alors = (Linstr) (pilesArbres.depiler());
    Expr ec = (Expr) (pilesArbres.depiler());
    pileArbres.empiler(new Si(ec, alors, null));
```

L'action (4) dépile trois arbres abstraits : tout d'abord l'arbre abstrait des instructions de la partie *Sinon* (rangées dans la variable `sinon`), puis l'arbre abstrait de la partie *Alors* (rangé dans la variable `alors`), puis enfin l'arbre abstrait de l'expression (rangé dans la variable `ec`). Elle termine par empiler le nouvel arbre d'instruction conditionnelle construit à partir de ces trois variables. L'action (5) fonctionne selon le même principe.

2.2.4 Arbre abstrait d'une expression

Il n'y a guère plus de problème pour la construction de cet arbre abstrait d'expression arithmétique binaire. Chaque action dépile autant d'arbres que nécessaires pour construire l'arbre résultat.

```
EXPR     -> EXPR OPBIN EXPR (6)
         |  OPERANDE
OPERANDE -> constanteEnt (7)
         |  ACCES
ACCES    -> ident (8)
         |  '+' (9)
         |  '-' (10)
         |  '*' (11)
         |  '/' (12)
         |  '==' (13)
```

avec les actions définies par

```
(6)  Expr opd = (Expr) (pilesArbres.depiler());
     Expr oper = (Expr) (pilesArbres.depiler());
     Expr opg = (Expr) (pilesArbres.depiler());
     oper.setGauche(opg);
     oper.setDroit(opd);
     pileArbres.empiler(oper);
(7)  Integer val = (Integer) (pilesArbres.depiler());
     pileArbres.empiler(new Nombre(val.value()));
(8)  String nom = (String) (pilesArbres.depiler());
     pileArbres.empiler(new Ident(nom));
(9)  pilesArbres.empiler( new Addition());
(10) pilesArbres.empiler( new Soustraction());
(11) pilesArbres.empiler( new Produit());
(12) pilesArbres.empiler( new Division());
(13) pilesArbres.empiler( new Egal());
```

L'action (6) permet d'enraciner les deux opérandes d'un opérateur binaire : l'opérande droit de l'opérateur est dépilé, rangé dans la variable `opd`, puis l'opérateur, rangé dans la variable `oper` et pour finir l'opérande gauche, rangé dans la variable `opg` ; ces deux opérandes sont enracinées à l'opérateur avant que celui-ci soit empilé à son tour. Les actions (9) à (13) créent un noeud correspondant à l'opérateur, sans aucun fils à ce moment là. Dans les actions (7) et (8), l'arbre empilé est crée à partir de l'unité lexicale, elle-même dépilée au préalable.

2.2.5 Exemple

Soit le code source suivant :

```
si (a == b) alors
    si (y == 0) alors x = 1 sinon x = 2 finsi
sinon
    si a alors x = 3 * x ; y = 0 finsi
finsi
```

L'objectif est de suivre le déroulement de l'analyse syntaxique et d'exécuter les actions de façon à construire l'arbre abstrait de la liste d'instructions. La figure 12 montre un extrait de l'arbre syntaxique du texte source, dans lequel les parties *Alors* et *Sinon* sont symbolisées par des triangles. Les actions sont exécutées au cours de l'analyse syntaxique, en fin de règle, au moment de la réduction. Commençons par exécuter les actions destinées à créer l'arbre abstrait de l'expression condition, c'est-à-dire les actions (8), (8) et (13).

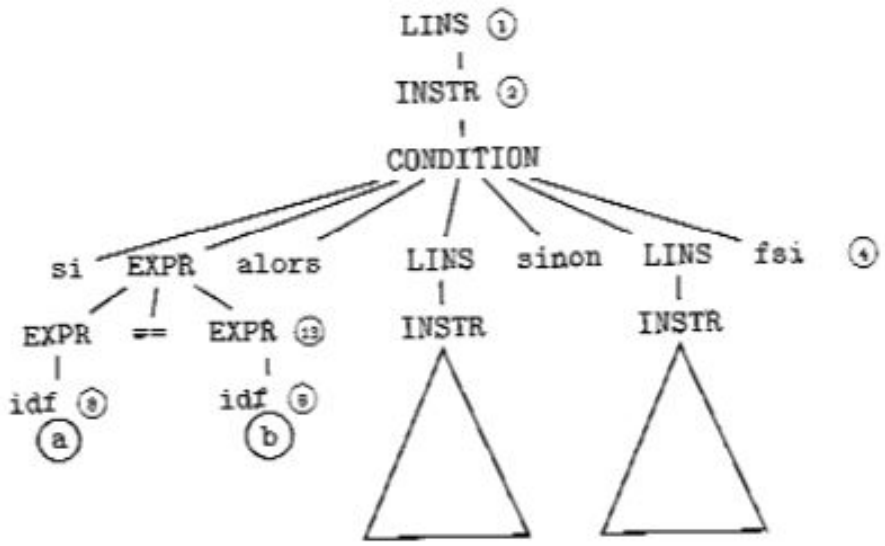

FIGURE 12 – Arbre syntaxique du programme source donné en exemple

```
(8)  String nom = (String) (pilesArbres.depiler());
     pileArbres.empiler(new Ident(nom);
(8)  String nom = (String) (pilesArbres.depiler());
     pileArbres.empiler(new Ident(nom);
(13) Expr opd = (Expr) (pilesArbres.depiler());
     Expr opg = (Expr) (pilesArbres.depiler());
     pilesArbres.empiler( new Egal(opg, opd));
```

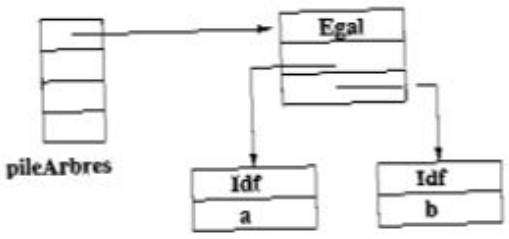

FIGURE 13 – Arbre abstrait de l'expression a==b disponible au sommet de la pile

A l'issue de l'exécution de ces actions, la pile des arbres `pileArbres` contient en son sommet l'arbre abstrait de la condition, comme l'illustre la figure 13. Vient ensuite l'exécution des actions de la partie *Alors* qui empilent successivement dans la pile des arbres `pileArbres`, l'arbre abstrait de la condition, puis au dessus, celui

de la partie *Alors* puis enfin l'arbre de la partie *Sinon*. La dernière action de la partie *Alors* (action (4)) a pour effet de dépiler les trois noeuds en sommet de pile pour les enraciner de façon à former l'arbre de la partie *Alors*, comme le montre la figure 14. Pour terminer, l'action (1) crée une liste réduite à une instruction.

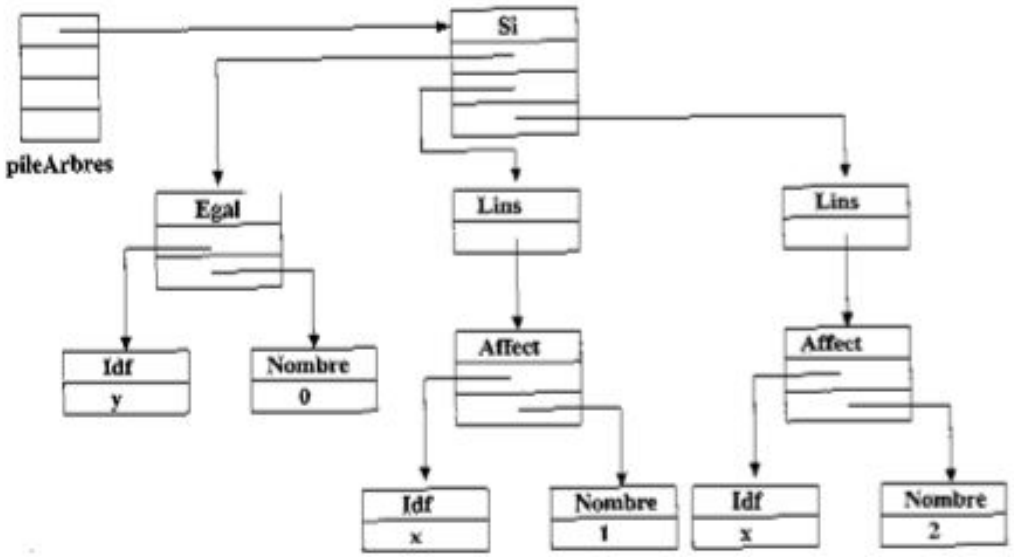

FIGURE 14 – L'arbre de la partie 'Alors' est au sommet de la pile

L'exécution des instructions de la partie 'sinon' produit un effet similaire et crée l'arbre abstrait de la partie *Sinon* de la première instruction conditionnelle, comme l'illustre la figure 15. Notons que dans cette figure, sous le sommet de pile se trouve encore l'adresse de l'arbre abstrait des instructions formant la partie *Alors*.

L'analyse syntaxique du texte se poursuit par une suite de réductions et l'exécution des actions (4), (2) et (1). Ces actions provoquent le dépilement des trois arbres de la condition pour les enraciner, puis créent et empilent un arbre d'une seule instruction accessible depuis le sommet de la pile.

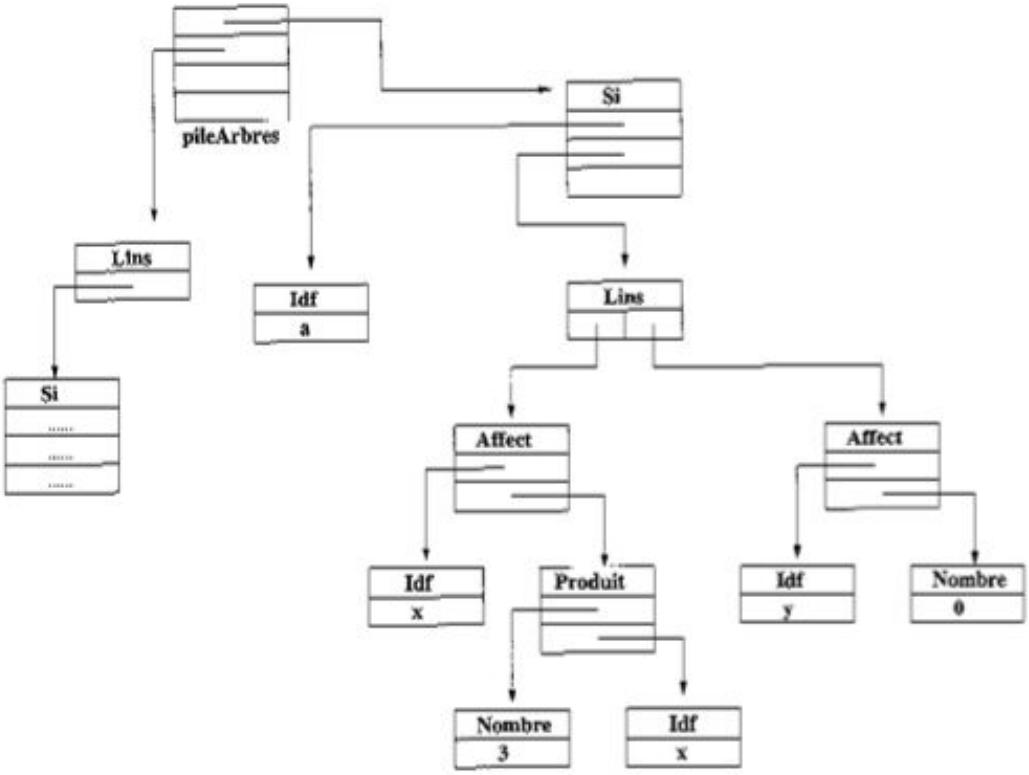

FIGURE 15 – L'arbre de la partie 'Sinon' est au sommet de la pile