ENSSAT
LANNION

UNIVERSITÉ DE RENNES 1

INFO2

Projet de compilation
CompNNO
Réalisation d'un compilateur
Pour le langage nilNovi Objet

Mars 2019

Damien Lolive

1. Présentation générale

Partant d'un analyseur syntaxique et lexical préexistant pour le langage NILNOVI OBJET étudié en cours, l'objectif du projet est de réaliser un système de développement permettant la production du code objet et son exécution. Sont également mis à disposition 1) l'interpréteur permettant l'exécution du code objet ; 2) un ensemble de méthodes permettant la construction et la consultation de la table des identificateurs (TDI). L'architecture générale de l'ensemble est présentée à la figure 1.

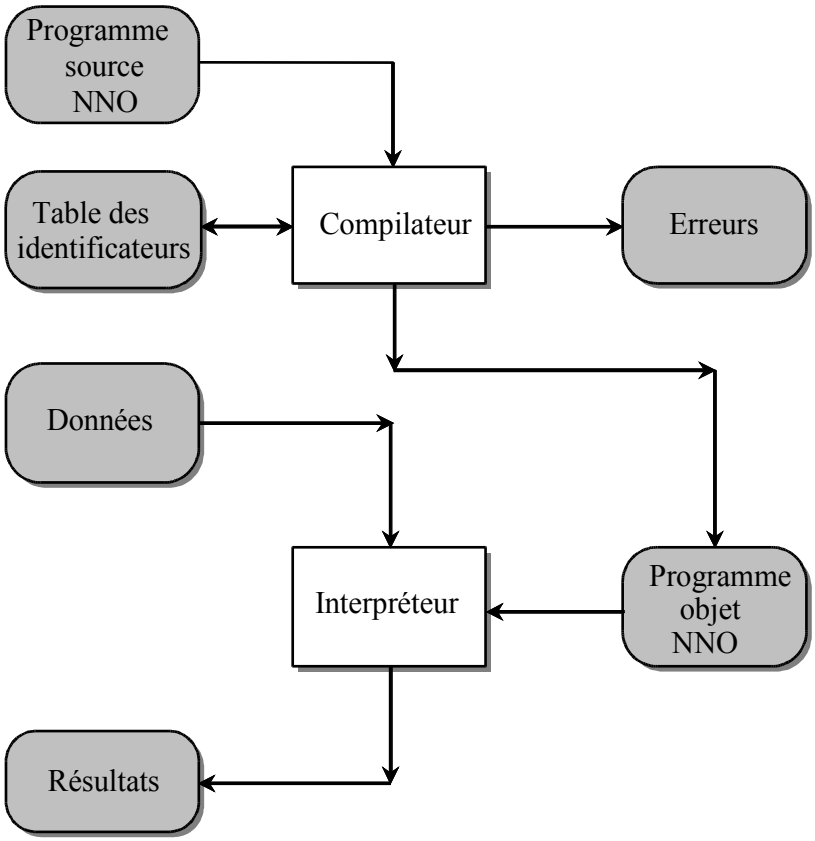


Figure 1. Architecture de l'ensemble

De manière plus précise, l'objectif du projet est de créer certains composants (analyseur sémantique/génération) et d'enrichir certains des composants fournis de façon à disposer d'un « système de développement » NILNOVI OBJET dont l'architecture détaillée est présentée à la Figure 2.

Un compilateur fonctionnel NILNOVI OBJET est disponible sous le système Linux (commande nilnovi). Son fonctionnement est conforme à l'interface décrite à la page 32, à l'exception du bouton « non affecté » qui, lorsqu'on clique dessus, délivre la liste des méthodes construisant la table des identificateurs invoquées pour la compilation réalisée (cf. § 1, interface *TListeEnt*).

En outre une version fonctionnelle sans trace est fournie.

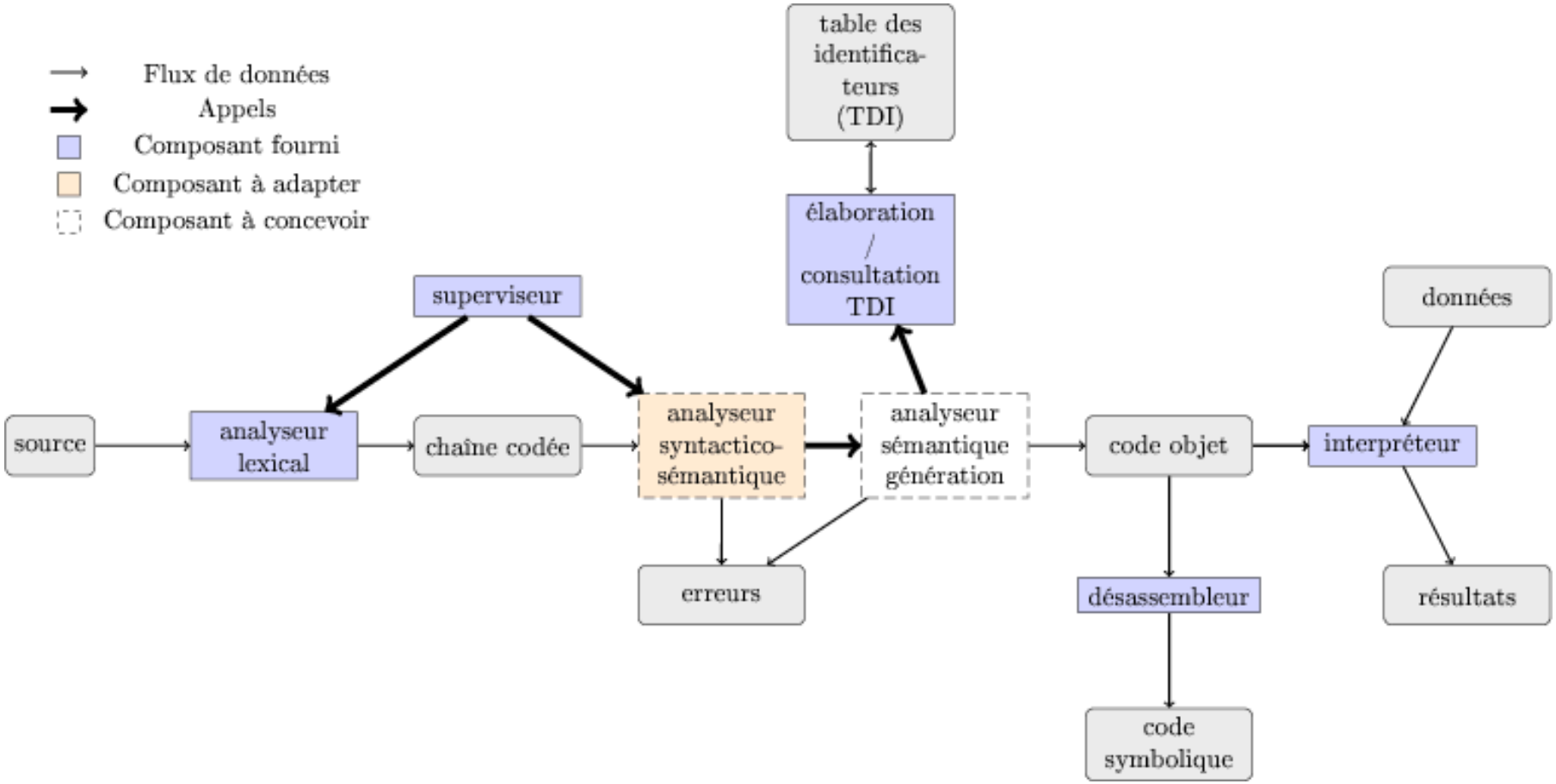


Figure 2 Synoptique du travail à réaliser

2. Le langage à compiler

Il s'agit du langage nilNovi Objet décrit dans [1]. Cette section vient compléter les éléments de définition présents dans le document sus-cité. On décrit ci-dessous les propositions qui doivent être avérées pour qu'un programme *syntaxiquement correct* soit *exécutable*.

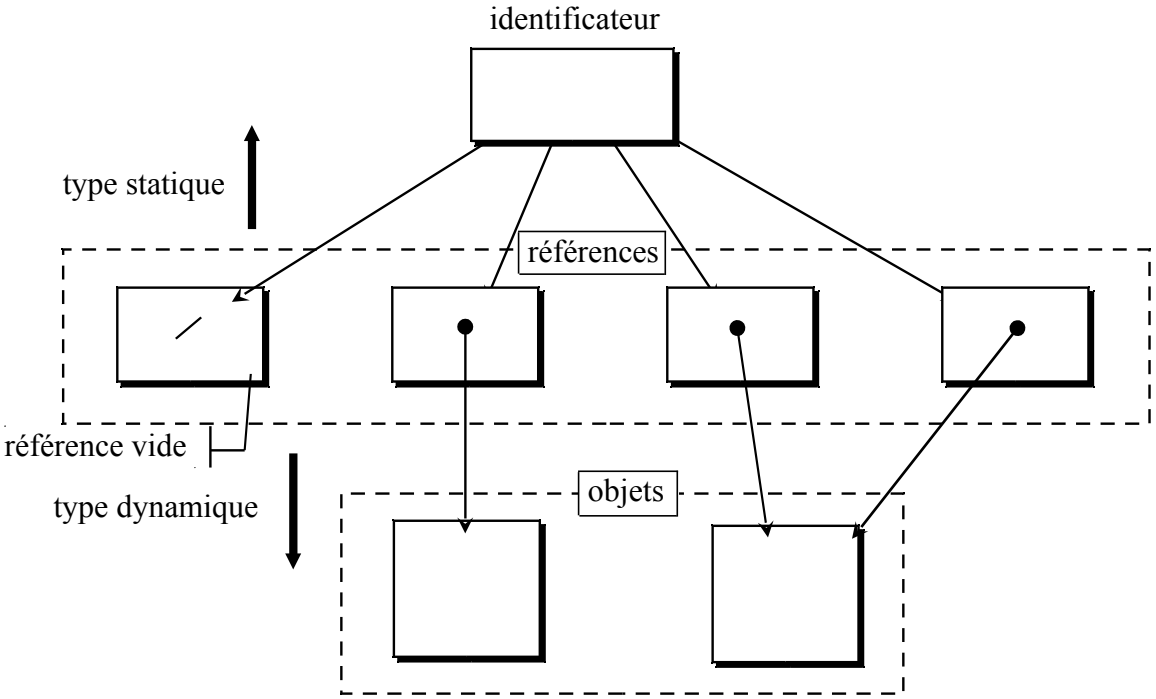
Nous avons vu en cours que la hiérarchie de classes dans un programme nilNovi Objet dote les classes d'une relation d'ordre partiel que l'on note ≤. Cette relation d'ordre s'étend naturellement aux types scalaires puisqu'elle se limite alors à l'égalité.

Définitions. À un identificateur d'objet (variable, paramètre ou attribut) est associé (à l'exécution) une ou plusieurs *références*. Une *référence* peut être dans l'état créé ou dans l'état vide. Dans le premier cas (état créé) la référence *désigne* un objet. Un objet donné peut être désigné par 0, 1,…, *n* références.

Définition. Le *type* d'un identificateur d'objet est le nom de la classe utilisée dans sa déclaration.

Définition. Le *type (statique)* d'une référence est le type de l'identificateur d'objet auquel il est associé.

Définition. Le *type dynamique* d'une référence est le type de l'objet qu'elle désigne.



Définition. *Type d'une expression d'objet.* Une expression d'objet réduite à un identificateur d'objet a comme type le type de cet identificateur. Une expression d'objet constituée par un appel (correct) à un constructeur est du type de la classe utilisée comme préfixe dans l'appel du constructeur. Une expression d'objet constituée par un appel à une méthode-fonction (délivrant un type classe) appliquée à une expression d'objet de type *e* est du type délivré par

la méthode-fonction pour le type *e*. Plus de précisions sont disponibles dans le document de cours.

1. Dans les expressions scalaires (arithmétiques ou booléennes) les opérandes et leurs opérateurs doivent être du même type. 12 étant du type entier, l'expression

   ```
   12+a=y or b
   ```

   est correcte si *a* et *y* sont de type entier et si *b* est de type booléen.

   ```
   5+true
   ```

   est une expression incorrecte.

2. Dans une affectation telle que

   ```
   a:=e
   ```

   où *a* est un identificateur de variable globale (déclaré dans la partie impérative du programme principal) ou bien est un identificateur de variable locale, d'attribut ou de paramètre de mode **in out**, le type de *e* doit être ≤ au type de *a*. Si *x* est une variable entière, l'expression

   ```
   x:=x+1
   ```

   est correcte puisque le type de l'expression est « integer » ainsi que le type de la variable affectée et donc *type*(*x*+1) ≤ *type*(*x*). Par contre l'instruction

   ```
   x:=true
   ```

   est incorrecte puisque les deux types (*integer* et *boolean*) n'entretiennent pas de liens à travers la relation d'ordre.

   Si *C* est une classe racine et *B* une classe fille de *C* et si par ailleurs *x* est une variable de type *C* et *y* une expression d'objet de type *B*

   ```
   x:=y
   ```

   est une affectation correcte puisque *type*(*y*) ≤ *type*(*x*).

3. Les conditions d'une boucle while ou d'une alternative sont des expressions *booléennes*. Le fragment de code

   ```
   while 13+3 loop
   ```

   est incorrect (l'expression est entière).

4. Les paramètres effectifs et les paramètres formels d'une opération doivent être compatibles en nombre, et mode (**in**, **in out**). De plus on doit avoir : *type*(*paramètre effectif*) ≤ *type*(*paramètre formel*).
5. Un paramètre *formel* et un paramètre *effectif* sont compatibles en *mode* si le premier est de mode **in** et le second est une expression, ou alors si le premier est de mode **in out** et le second une variable, un attribut, ou un paramètre de mode **in out**.
6. Le paramètre formel des procédures prédéfinies *error*, *get* et *put* est de type *integer* et de mode respectif **in**, **in out** et **in**.
7. Une instruction **return** ne peut apparaître que dans une méthode-fonction. Mais une méthode-fonction doit obligatoirement être dotée d'au moins une instruction **return**[1].
8. Soit *r* le type délivré par une méthode-fonction et *e* l'expression qui suit une instruction **return** dans la méthode-fonction. On doit avoir *type*(*e*) ≤*r*.
9. Dans l'appel d'un constructeur *a.c*(…), *a* est un identificateur de classe et *c* est le constructeur de la classe *a*.

[1] Ceci constitue une vérification grossière puisqu'elle ne permet pas de s'assurer que l'exécution d'une méthode-fonction s'achève bien par l'exécution d'un return. Une vérification plus élaborée serait possible.

10. Dans un appel de méthode *o.m*(…), *o* est une expression d'objet (qui à l'exécution désignera un objet effectif) et *m* est une méthode du type de *o*.
11. Dans une classe non racine, une méthode virtuelle redéfinie a obligatoirement le même profil que la méthode qu'elle redéfinit.
12. Toutes les méthodes déclarées dans la rubrique **interface** d'une classe doivent être définies dans la rubrique **implementation** de cette même classe. Toutes les méthodes définies dans la rubrique **implementation** d'une classe doivent avoir été déclarées dans la rubrique **interface** de cette même classe.
13. Un appel de méthode-procédure a le statut d'une *instruction*. Un appel à une méthode-fonction ou à un constructeur a le statut d'une *expression*.
14. Un identificateur de classe ou de variable globale ne peut être doublement défini.

```
type a is class
  ...
end;
a:integer ;
```

est erroné : l'identificateur *a* est doublement défini.

15. Un identificateur d'attribut ne peut être doublement défini dans une même hiérarchie de classes.

```
type a is class
  x: integer ;
  ...
end;
type b is class(a)
   x: integer ;
   ...
end;
```

est erroné : l'identificateur d'attribut *x* est doublement défini dans la hiérarchie de la racine *a*.

16. Un identificateur de paramètre formel ou de variable locale ne peut être doublement défini dans une même opération.

```
function f1(a:integer) return integer;
...
function f1    is
    a:integer;
```

est erroné : *a* est défini comme paramètre et comme variable locale.

17. Un identificateur de méthode ne peut être doublement défini dans une même classe.
18. Tout identificateur utilisé doit avoir été déclaré (la déclaration apparaît avant l'utilisation). Exceptions notables : les identificateurs d'opérations prédéfinies *get*, *put*, *error*, l'identificateur « *this* », les types « *integer* » et « *boolean* ». Un identificateur de classe est réputé défini dès l'occurrence du mots clé **class** dans le cas d'une classe racine ou dès l'occurrence de la «) » spécifiant la classe dont elle hérite dans le cas d'une classe fille. Cette convention permet 1) d'éviter qu'une classe hérite d'elle même, 2) d'autoriser une classe à posséder des champs qui sont du type de cette classe.
19. Une classe doit posséder un et un seul constructeur.

20. Un paramètre formel de mode **in** a le statut d'une expression. Un paramètre formel de mode **in out** a le statut d'une variable (en particulier il peut être utilisé comme paramètre effectif de mode **in out**).

21. Dans une classe, la définition d'un constructeur (resp. méthode-procédure, méthode-fonction) doit correspondre à la déclaration d'un constructeur (resp. méthode-procédure, méthode-fonction).

```
constructor f1();   //déclaration de f1
   ...
function f1() is    //définition de f1
```

est erroné : un constructeur est déclaré, une fonction est définie.

22. L'identificateur *this* joue sémantiquement le rôle d'un attribut *constant* qui désignerait l'objet considéré. En conséquence 1) il est interdit d'affecter cet identificateur, 2) *this* ne peut être utilisé en dehors d'une classe.

23. Les types *integer* et *boolean* sont des types scalaires et non des types classe. Aucune classe ne peut donc hériter d'eux.

3. L'analyseur lexical et syntaxique

Le développement du projet se fait en Java à partir d'un analyseur syntaxique ad hoc LL(1) (par descente récursive). L'annexe 1 présente en particulier les programmes d'analyse lexicale et syntaxique fournis.

4. Analyse sémantique et production du code objet

La fonction d'analyse sémantique a pour rôle d'effectuer les contrôles extra-syntaxiques (cf. §2). Elle s'appuie principalement sur la TDI (cf. Figure 2 et § 6) qu'elle construit et consulte. Ainsi que nous l'avons vu en cours, la fonction d'analyse sémantique est mise en œuvre de deux façons complémentaires :

1. en modifiant les méthodes de l'analyseur syntaxique afin qu'elles véhiculent et calculent la valeur des attributs sémantiques,

2. en insérant dans l'analyseur syntaxique des appels à des méthodes effectuant véritablement le contrôle sémantique. Ces méthodes consultent en général la TDI, elles ont également pour rôle de *construire* la TDI.

La production du code objet a pour objectif de construire le programme objet qui sera exécuté par l'intermédiaire de l'interpréteur. Le langage NILNOVI OBJET est conçu pour réduire le nombre et la difficulté des reprises (c'est-à-dire des retours sur le programme objet nécessaires pour compléter une instruction par une information qui n'était pas disponible au moment de la création de l'instruction). Certaines reprises (celles liées aux structures de contrôle, et aux appels de méthodes-procédures) peuvent être gérées par la pile (Java) de l'analyseur syntaxique. D'autres (celles liées aux sauts au-dessus de la définition d'une classe) sont gérées par l'intermédiaire de la table des identificateurs. La fonction de production de code objet est mise en œuvre soit en adaptant les méthodes effectuant l'analyse sémantique soit en utilisant des méthodes spécifiques, selon les cas.

5. Interpréteur

L'interpréteur permet l'exécution du code objet produit par le compilateur. Son code Java fait partie de la fourniture (cf. annexe 1). Afin de faciliter la phase de mise au point un désassembleur[2] accompagne la fourniture.

6. Table des identificateurs

Il est en général impossible de compiler correctement un programme en n'utilisant que les informations fournies par les analyseurs lexical et syntaxique. Ainsi par exemple dans nilNovi Objet lorsque l'on compile une affectation de variable dans la partie impérative du programme, on a besoin de connaître l'adresse statique de cette variable. Et pourtant cette information ne peut être déduite de la forme du symbole qui identifie cette variable. Elle dépend de ce qui précède dans le programme source : nombre de classes et de variables déclarées. Ce type d'information doit être collecté et organisé pendant la compilation afin d'être disponible au moment voulu.

La structure qui enregistre ces informations est la table des identificateurs (TDI). Elle joue un rôle central dans le processus de compilation tant pour permettre la production d'un code objet correct que pour détecter les incorrections qui ne sont pas découvertes par l'analyse syntaxique (ainsi par exemple une double déclaration d'une variable globale sera détectée en consultant la TDI).

En nilNovi Objet, la TDI est une suite d'entités (classes et variables globales). Afin de vous permettre de vous focaliser sur les aspects plus spécifiques de la compilation, les méthodes permettant la construction et la consultation de la TDI sont fournies (cf. annexe 1). Il vous suffira de placer des appels à ces méthodes dans les méthodes d'analyse sémantique sans avoir à vous préoccuper de la structure intime de la TDI.

7. Travail à réaliser

En partant des programmes sources Java fournis, il s'agit de concevoir, développer et mettre au point le système de développement nilNovi Objet.

On proposera une démonstration significative. Un dossier de programmation sera rédigé. Ce dossier comprendra en particulier :

- un rapport de projet qui développe les points suivants :
    - analyse/programmation du problème posé et sources commentés des différentes unités développées,
    - description succincte des jeux d'essai originaux (s'il y a lieu),
    - répartition des tâches entre les membres du groupe,
    - évaluation du nombre d'heures passées aux différentes phases (analyse, programmation, mise au point, rédaction du rapport),
    - état du programme (bogues éventuelles, parties non testées) et évolutions possibles.

[2] Un désassembleur produit le code symbolique à partir du code objet.

— un dossier de maintenance fournissant notamment la liste des différentes unités et leurs dépendances ainsi que les différentes commandes permettant d'obtenir un exécutable,

— des fiches de tests pour nilNovi Algorithmique et nilNovi Objet (cf. les fiches à remplir à la fin du document). Dans le cas où votre compilateur n'atteint pas la zone de texte où est localisée la caractéristique à mettre en évidence, il vous appartient, dans la mesure du possible, d'aménager le programme nilNovi de façon à y parvenir.

L'ensemble des programmes sources sera fourni à l'issue du projet. Il sera accompagné d'un fichier de type « readMe » décrivant le mode d'emploi.

Annexe 1
NilNovi en Java

Cette annexe présente les programmes sources Java fournis à chaque groupe au début du projet. Il s'agit principalement de l'analyseur lexical, de l'analyseur syntaxique, de l'interpréteur et de l'interface Java permettant la gestion de la TDI. Dans la première partie nous décrivons les principes architecturaux, dans la seconde partie nous formulons quelques remarques à propos des systèmes de développement susceptibles d'être utilisés. La troisième partie décrit l'organisation (répertoires/sous-répertoires) des programmes sources fournis. Les quatrième, cinquième, sixième et septième parties décrivent respectivement l'analyseur lexical, l'analyseur syntaxique, l'interpréteur et l'interface Java de la TDI. La huitième partie précise l'interface entre les programmes fournis et le développement à réaliser. Enfin la neuvième partie décrit l'interface graphique fournie.

1. Introduction

La Figure 2 montre comment s'articulent les modules fournis (analyseur lexical, interpréteur, désassembleur et interface TDI), le module à adapter (analyseur syntaxique) et le module à concevoir (analyseur sémantique/génération).

2. Système de développement

Pour le développement Java vous pouvez utiliser l'IDE Eclipse. Ci-dessous la séquence de commandes Eclipse pour effectuer un première exécution de la fourniture

```
Fenêtre Java – Eclipse SDK (fenêtre Eclipse)
  File/Import
    Fenêtre Import s'ouvre
      Choisir General → Existing projects into workspace
      bouton Next
        Sélectionner « Select Archive File »
          bouton Browse :
            fenêtre rechercher un dossier s'ouvre
              sélectionner l'archive du projet(srcTDIEclipse-10-11.zip ici)
              bouton OK
          bouton Finish
            (Eclipse passe un certain temps à construire l'espace de
             travail)
  s'assurer que le volet package explorer de la vue package est sélectionné
    afficher les composants du projet (s'assurer qu'il existe un package
      nommé (default package))
      afficher les programmes XXX.java de ce default package. L'un d'eux
       est le main (ici nnoClass.java)
      cliquer avec le bouton droit sur le nom du main (nnoClass.java)
        menu déroulant s'ouvre
          cliquer sur Run as/ 2. Java Application. Le programme se lance
```

3. Organisation de la fourniture

3.1 Description des répertoires

nno : répertoire racine. Ce répertoire se décompose en 6 répertoires fils.

1. *bin* : répertoire par défaut créé par Eclipse pour mettre les .class compilés, de manière à séparer sources et exécutables.
2. *documentation* : répertoire contenant la documentation en ligne pour l'interface Java *PListeEnt.java* (fichier source non disponible). Cette documentation porte sur les méthodes permettant la création et la consultation de la TDI. Les fichiers et sous-répertoires de ce répertoire sont produits automatiquement par le générateur de documentation javadoc. Son contenu ne présente pas d'intérêt direct pour le projet. Son utilisation se fait par l'intermédiaire du fichier *srcTDIEclipse-10-11/PListeEnt.html* (cf. §3.2, point 1.).
3. *jeuxEssaiNNA* : ce répertoire contient l'ensemble des jeux d'essai proposés de type nilNovi Algorithmique.
4. *jeuxEssaiNNO* : ce répertoire contient l'ensemble des jeux d'essai proposés de type nilNovi Objet.
5. *src :*
    1. *analyseurLexical* : répertoire contenant l'ensemble des classes pour l'analyse lexicale et pour la production de messages d'erreurs. Ce répertoire comprend trois sous-répertoires :
        1. *tableMotsCles* : répertoire contenant le fichier qui permet l'élaboration et l'exploitation de la table des mots clés du langage nilNovi Objet.
        2. *uniteLexicale* : répertoire contenant les classes qui permettent de représenter les unités lexicales constituant la chaîne codée.
        3. *MesExceptions* : répertoire pour la gestion d'une exception qui permet d'afficher la position d'une erreur de compilation nilNovi.
    2. *analyseurSyntaxique* : répertoire contenant l'analyseur syntaxique proprement dit.
    3. *InterpreteurNNO* : répertoire contenant l'interpréteur du langage nilNovi. Ce répertoire est constitué de deux sous-répertoires :
        1. *instructions* : ce répertoire contient un fichier par instruction machine nilNovi. Chaque fichier décrit le code machine de l'instruction et la façon de désassembler l'instruction.
        2. *programmeObjetNNO* : ce répertoire contient l'interpréteur nilNovi.
    4. *nno* : ce répertoire contient les fichiers liés à l'interface fenêtrée.
    5. *ptGen* : ce répertoire contient des exemples de points de génération.
6. Lib :
    1. *TListe* répertoire contenant l'unique sous-répertoire *ptGen* de gestion de la TDI.
        1. *ptGen* : répertoire contenant l'unique sous-répertoire *TListeEnt* de gestion de la TDI. *TListeEnt* contient lui-même 4 sous-répertoires :
            1. *TEnt* (représentation des entités – classes et entités scalaires – dans un programme nilNovi Objet),
            2. *TListeEntScal* (représentation des entités scalaires – variables globales, attributs, paramètres, variables locales des méthodes),
            3. *TListeMethVirt* (représentation des méthodes virtuelles),
            4. *TMeth* (représentation des méthodes)

    Ces quatre sous répertoires ne contiennent que des fichiers .class (les fichiers .java ont été rendu inaccessibles car inutiles : la gestion de la TDI se fait uniquement à travers

l'interface *PListeEnt.class* décrite au § 7 et 8. Ces fichiers sont protégés et ne doivent pas être détruits.

3.2 Description des fichiers dans les répertoires

Sauf exception, les fichiers *.class (correspondant à la compilation de fichiers *.java) ne sont pas décrits.

1. **src** :
   - *nnoClass.java* : fichier contenant la méthode *main*.
2. **src/analyseurLexical** :
   - *anaLex.java* : classes Java qui effectuent la lecture du fichier source et qui produisent la chaîne codée constituée des unités lexicales reconnues dans le texte source. L'analyse lexicale est effectuée conformément à sa spécification sous la forme d'un langage d'états finis (cf. §4 ci-dessous).
3. **src/analyseurLexical/tableMotsCles** :
   - *tableDesMotsCles.java* : contient la classe pour la gestion des mots clés du langage NILNOVI.
4. **src/analyseurLexical/mesExceptions** :
   - *monException.java* : classe qui délivre la position linéaire permettant d'identifier les erreurs de compilation dans l'éditeur.
5. **src/analyseurLexical/uniteLexicale** :
   - *el.java* : classe abstraite java pour la représentation des unités lexicales. Cette classe est instanciée en 5 classes spécialisées selon la nature de l'unité lexicale.
   - *caractere.java* : classe pour la représentation des unités lexicales qui sont des caractères (par exemple ';' dans « **end** ; »).
   - *entier.java* : classe pour la représentation des unités lexicales qui sont des entiers (comme 1543 dans l'affectation NILNOVI OBJET « x := 1543 »).
   - *fel.java* : classe pour la représentation de l'unité lexicale fictive « fin de la chaîne codée ». Un objet de cette classe joue un rôle de sentinelle dans l'analyse syntaxique.
   - *identificateur.java* : classe pour la représentation des unités lexicales qui sont des identificateurs (comme « *x* » et « *integer* » dans la déclaration NILNOVI OBJET « *x* : *integer* ; »).
   - *motCle.java* : classe pour la représentation des unités lexicales qui sont des mots clés NILNOVI OBJET (comme « *type* », « *is* » et « *class* » dans l'en-tête NILNOVI OBJET « **type** cl **is class** »).
6. **src/analyseurSyntaxique** :
   - *analyseurLL1.java* : analyseur syntaxique qui réalise une analyse LL(1) de la chaîne codée produite par l'analyseur lexical par la méthode de la descente récursive. Ce fichier devra être modifié pour transformer certaines des « procédures » de l'analyseur en « fonctions » délivrant un résultat qui sera exploité pour traiter les aspects sémantiques de la compilation.
7. **src/documentation** (et sous-répertoires) :
   fichiers de documentation produits par « javadoc » sur les 97 méthodes permettant la création et l'exploitation de la TDI (cf. §8).
8. **src/interpreteurNNO/instructions** :
   - *InstInstr.java* : classe abstraite pour la description des instructions machine sans argument.

- *InstInstr1.java* : classe abstraite pour la description des instructions machine avec un argument.
- *InstInstr2.java* : classe abstraite pour la description des instructions machine avec deux arguments.
- *instXX*.java : (XX est instancié par le nom des instructions machine du langage nilNovi). Chaque classe décrit le code machine de l'instruction ainsi que la façon de désassembler l'instruction.

9. **src/interpreteurNNO/programmeObjetNNO** :
   - *interpreteur.java* : fichier contenant la classe qui simule la machine nilNovi (voir [1] pour la sémantique de chaque instruction machine).
10. **src/jeuxEssaiNNA** : répertoire comprenant des jeux d'essai pour nilNovi Algorithmique. Ce répertoire contient deux types de fichiers nilNovi :
    - *erreurXX.nno* : fichiers nilNovi contenant un programme correct syntaxiquement mais erroné du point de vue sémantique. Votre compilateur devra détecter l'erreur.
    - *correctXX.nno* : fichiers nilNovi contenant des programmes corrects du point de vue syntaxique et sémantique et dont l'exécution produit un résultat.
11. **src/jeuxEssaiNNO** : répertoire comprenant des jeux d'essai pour nilNovi Objet. Ce répertoire contient deux types de fichiers nilNovi :
    - *erreurXX.nno* : fichiers nilNovi contenant un programme correct syntaxiquement mais erroné du point de vue sémantique. Votre compilateur devra détecter l'erreur.
    - *correctXX.nno* : fichiers nilNovi contenant des programmes corrects du point de vue syntaxique, sémantique et dont l'exécution produit un résultat.
12. **src/nno** : ce répertoire contient les fichiers permettant la gestion de l'interface. L'essentiel du contenu de ces fichiers a été produit automatiquement par l'IDE Jbuilder. Ces fichiers n'ont pas à être modifiés ni même consultés. Il contient également les fichiers image (*.gif) pour les boutons de l'interface fenêtrée.
13. **src/ptGen** :
    - *PointsGeneration.java* : exemple de points de génération. Cet exemple permet l'affichage en fin d'analyse des identificateurs des classes et des variables globales du programme nilNovi Objet analysé (cf. §8 pour plus de détails)
14. **lib/TListe/ptGen/TListeEnt** :
    - *TListeEnt.class* : fichier objet (protégé) mettant en œuvre la gestion de la TDI (cf. § 7).
    - *PListeEnt.class* : interface (au sens java) donnant accès aux méthodes du fichier *TListeEnt.class* mettant en œuvre la TDI. Les méthodes de cette interface sont décrites au §7. Elles permettent de créer et d'exploiter la TDI directement à partir de l'analyseur syntaxique ou dans les points de génération (ceux-ci restent à écrire, ils constituent l'essentiel du travail demandé dans ce projet). Ce fichier est protégé.

Les programmes fournis sont tels que lors de l'analyse d'un programme nilNovi Objet incorrect syntaxiquement, l'exécution se termine dès la première erreur détectée. Cette erreur est signalée dans le code source et un message de diagnostic est produit. La « compilation »[3] d'un programme nilNovi Objet fourni simplement (à titre d'exemple) la liste des identificateurs globaux présents dans le programme. Une demande d'exécution conduit à exécuter le programme objet standard créé par l'appel au point de génération *genCode* (fichier *src/PtGen/PointsGeneration.java*). Il s'agit d'un exemple décrit dans le polycopié de cours, celui portant sur les expressions arithmétiques.

[3] L'usage de ce terme est abusif tant que votre projet n'est pas achevé…

4. Analyseur lexical

Le rôle de l'analyseur lexical présenté ici est, partant d'un programme source nilNovi Objet, de construire une structure appelée « chaîne codée ». Alors qu'un programme source est constitué d'une suite de caractères, une chaîne codée est une liste d'éléments lexicaux (comme les entiers, les mots clés, les identificateurs, certains caractères particuliers) où les caractères ont perdus leur individualité et peuvent avoir été regroupés (comme dans un identificateur) ou même ont disparus (comme dans la représentation d'un entier).

L'analyseur lexical de nilNovi Objet a également comme rôle annexe 1) de lire le programme source depuis un fichier, 2) de faire disparaître les éléments inutiles à l'analyse syntaxique comme les espaces, les tabulations et les commentaires.

nilNovi Objet distingue 5 types d'unités lexicales. Chaque type est modélisé par une classe, elles ont en commun les numéros de ligne et de colonne où débute l'unité en question.

1. *entier* : contient la représentation binaire de l'entier considéré.
2. *identificateur* : contient la chaîne constituant l'identificateur.
3. *motCle* : contient la chaîne de caractères constituant le mot clé.
4. *caractere* : pour les unités qui ne sont ni des entiers ni des identificateurs ni des mots clés, contient le caractère considéré.
5. *fel* : unité fictive achevant la chaîne codée.

Exemple.

Le programme nilNovi ci-dessous

```
procedure pp is
  //un exemple
  i,som: integer;
begin
  som:=0;
  i:=23
end.
```

produira la chaîne codée suivante :

Numéro	Type	contenu
1	Mot clé	procedure
2	Identificateur	pp
3	Mot clé	is
4	Identificateur	i
5	Caractère	,
6	Identificateur	som

7	Caractère	:
8	Identificateur	integer
9	Caractère	;
10	Mot clé	begin
11	Identificateur	som
12	Mot clé	:=
13	Entier	0
14	Caractère	;
15	Identificateur	i
16	Mot clé	:=
17	Entier	23
18	Mot clé	end
19	Caractère	.
20	fel	

La production de la chaîne codée se fait à partir d'un analyseur d'états finis qui est codé sous la forme du diagramme syntaxique présenté à la Figure 3. Des traitements non syntaxiques[4] sont effectués dès que le type d'unité lexicale est identifié. Lorsqu'une unité lexicale est reconnue, elle vient allonger la chaîne codée. Ainsi par exemple on utilise l'algorithme connu sous le nom de « schéma de Horner » pour transformer la *représentation* décimale d'un entier en un entier « machine ». La distinction entre un identificateur et un mot clé se fait en utilisant la table des mots clés.

La chaîne codée est ensuite traitée par l'analyseur syntaxique. Les différents éléments de la chaîne codée seront accessibles dans les points de génération que vous allez rédiger. Nous verrons comment au § 8. Cette possibilité est indispensable pour réaliser effectivement le compilateur.

[4] Destinés à élaborer la représentation de l'unité lexicale considérée.

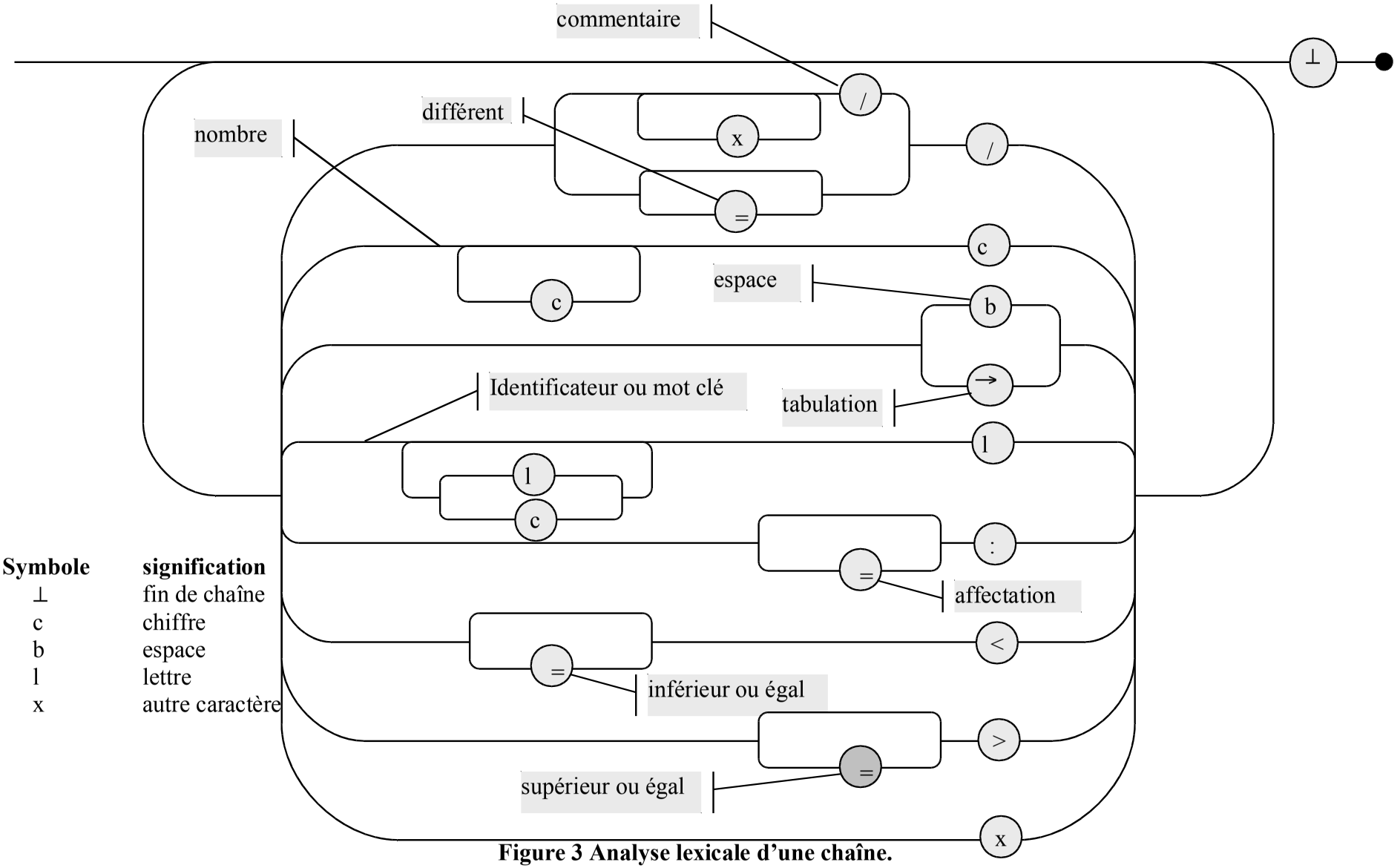
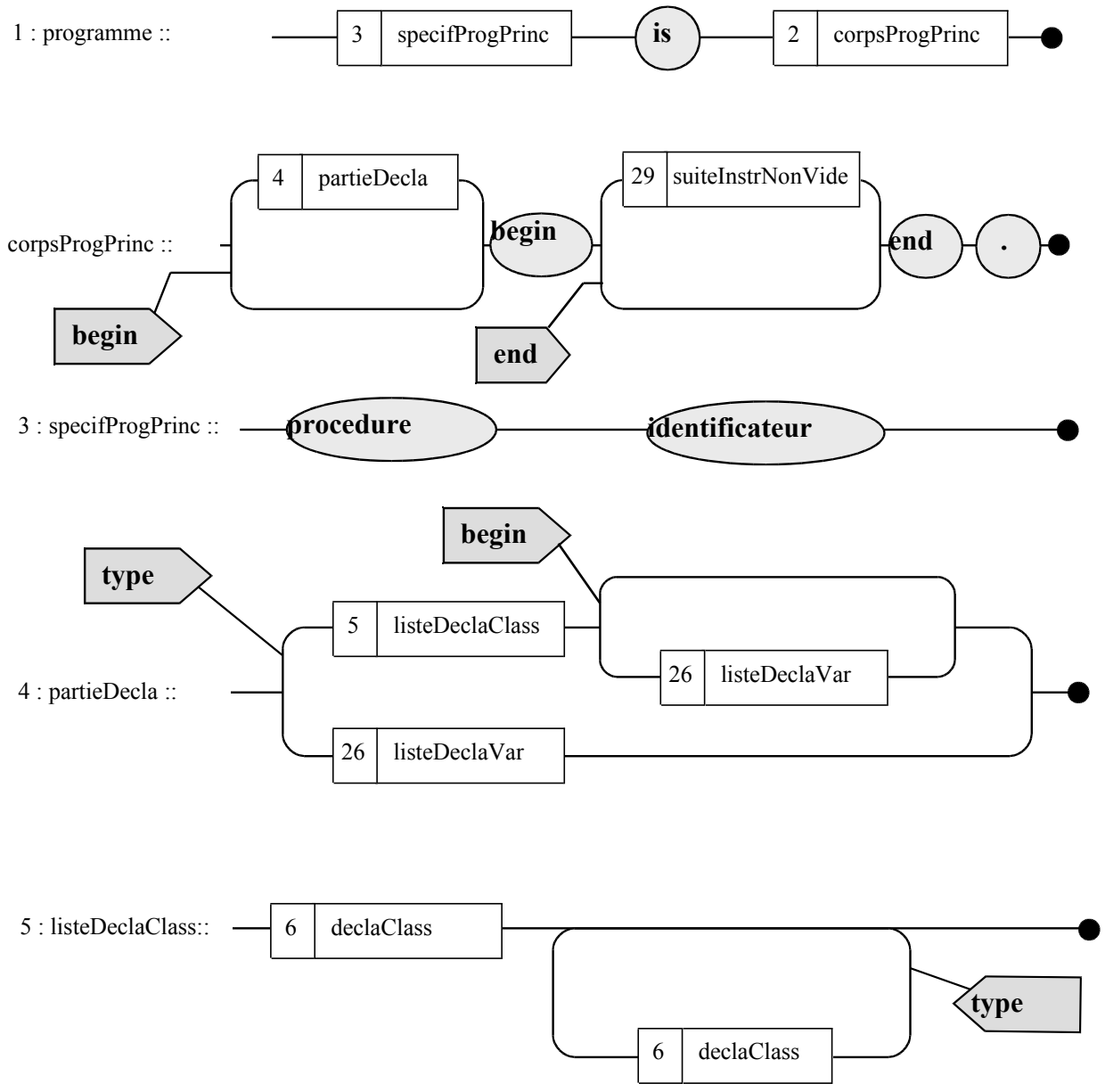
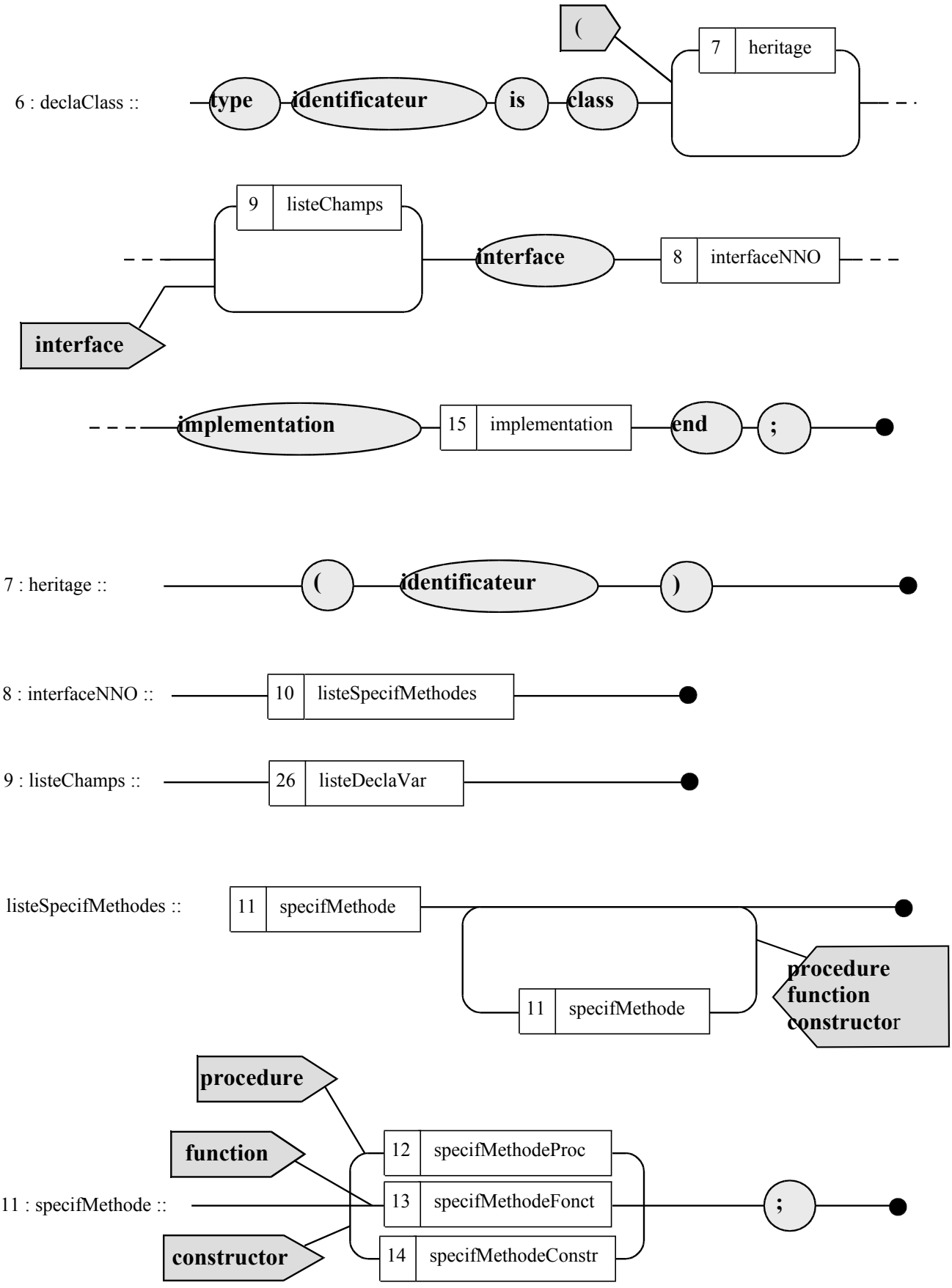


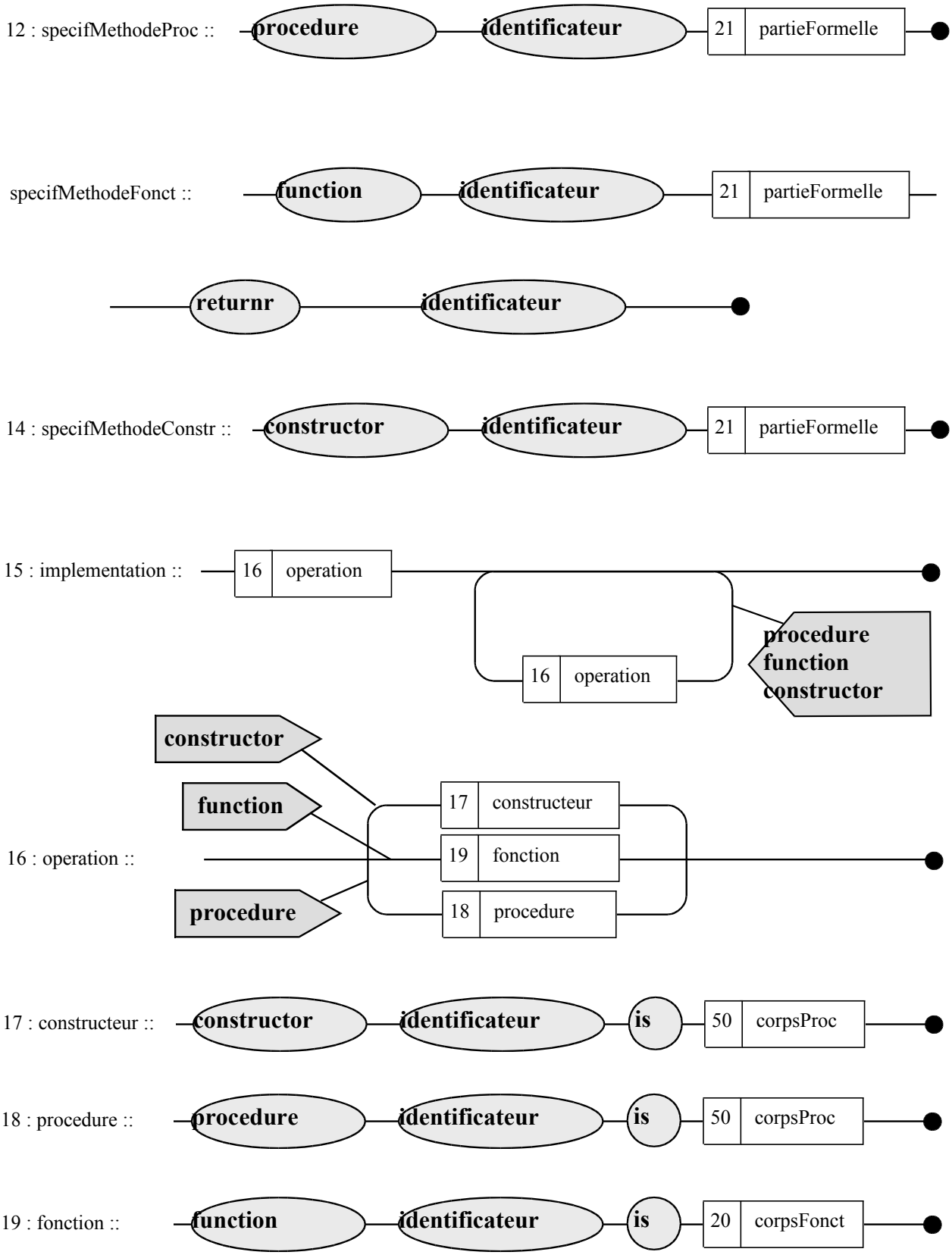
Figure 3 Analyse lexicale d'une chaîne.

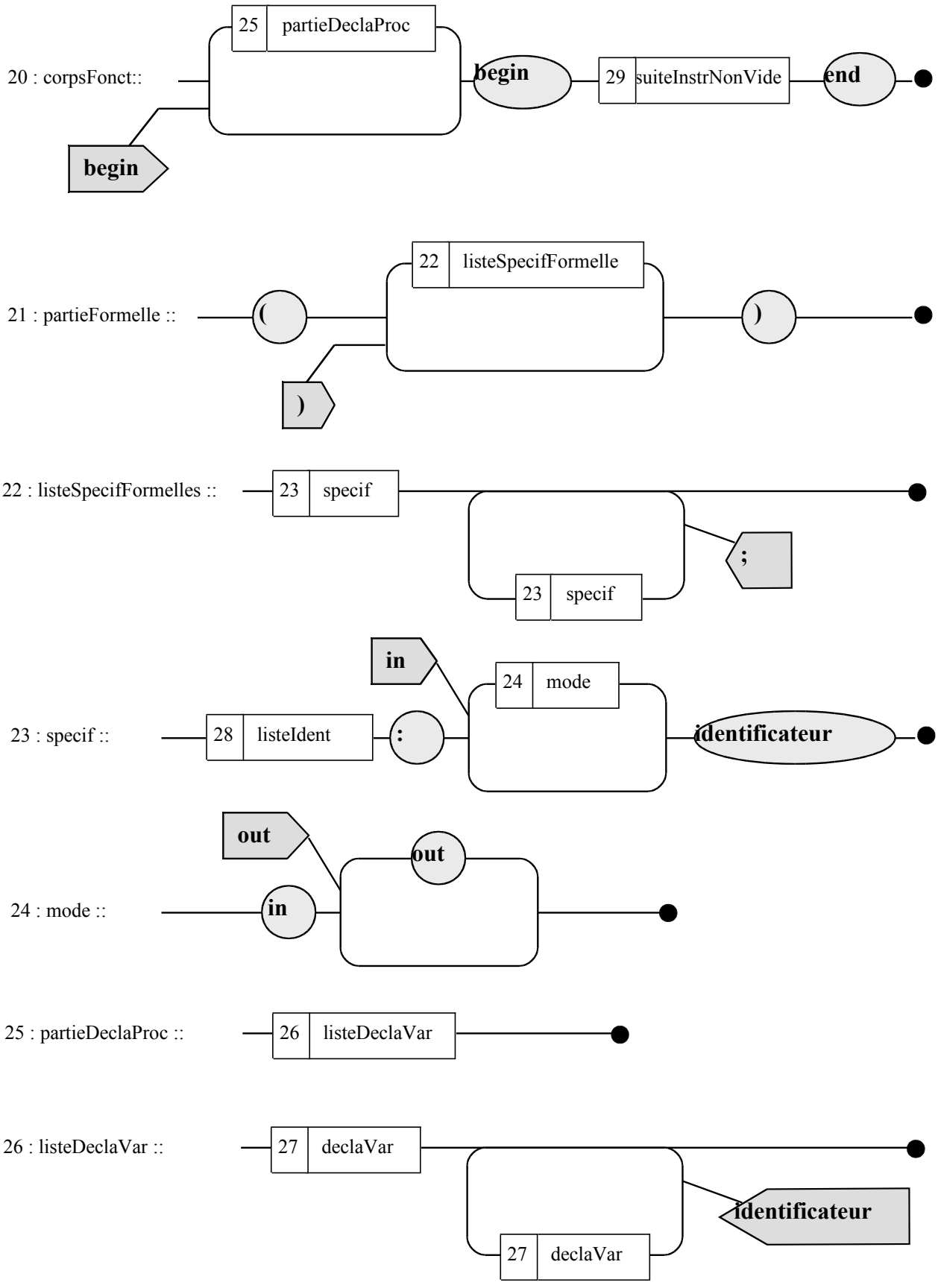
5. Analyseur syntaxique

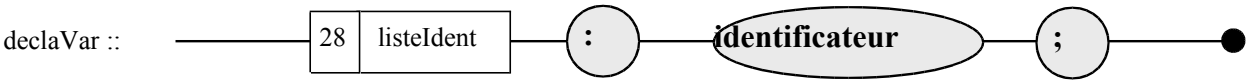
Pendant le cours nous avons étudié la grammaire NILNOVI OBJET. Cette grammaire est à l'origine d'un analyseur (codé en Java) qui vous est fourni. La grammaire du cours est une grammaire LR(1). Cette grammaire a été « raffinée » en grammaire LL(1) afin de faciliter son codage. Cette grammaire LL(1) a elle-même été « raffinée » en diagrammes syntaxiques : la récursivité droite à été supprimée au profit de « boucles ». Des « panneaux d'aiguillage » ont été apposés pour vérifier et exploiter le caractère déterministe de la description. Ces diagrammes ont alors été codés en Java (fichier *srcTDI-06-07/analyseurSyntaxique/analyseurLL1.java*). Ils sont présentés à la Figure 4.

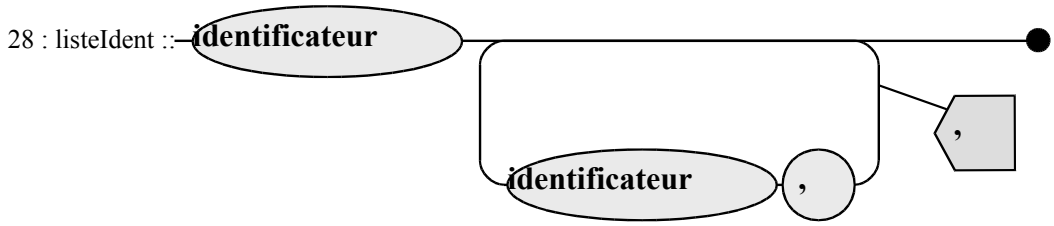


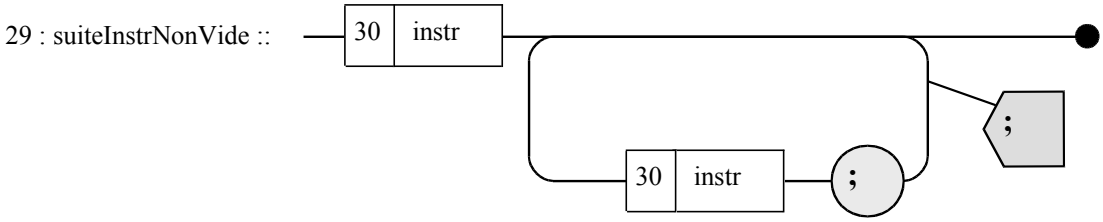

6 : declaClass ::
type
identificateur
is
class
(
7
heritage
9
listeChamps
interface
8
interfaceNNO
interface
implementation
15
implementation
end
;
7 : heritage ::
(
identificateur
)
8 : interfaceNNO ::
10
listeSpecifMethodes
9 : listeChamps ::
26
listeDeclaVar
listeSpecifMethodes ::
11
specifMethode
11
specifMethode
procedure
function
constructor
procedure
function
constructor
11 : specifMethode ::
12
specifMethodeProc
13
specifMethodeFonct
14
specifMethodeConstr
;

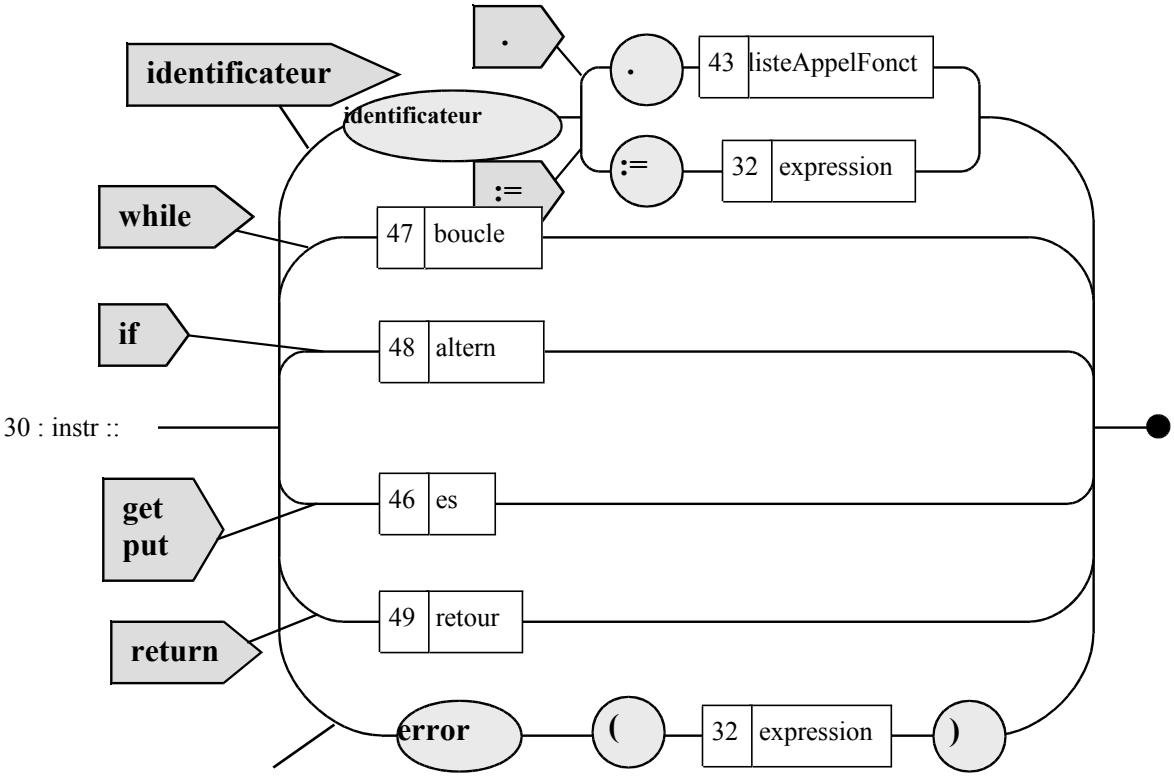

12 : specifMethodeProc ::
procedure
identificateur
21
partieFormelle
specifMethodeFonct ::
function
identificateur
21
partieFormelle
returnr
identificateur
14 : specifMethodeConstr ::
constructor
identificateur
21
partieFormelle
15 : implementation ::
16
operation
16
operation
procedure
function
constructor
16 : operation ::
constructor
function
procedure
17
constructeur
19
fonction
18
procedure
17 : constructeur ::
constructor
identificateur
is
50
corpsProc
18 : procedure ::
procedure
identificateur
is
50
corpsProc
19 : fonction ::
function
identificateur
is
20
corpsFonct

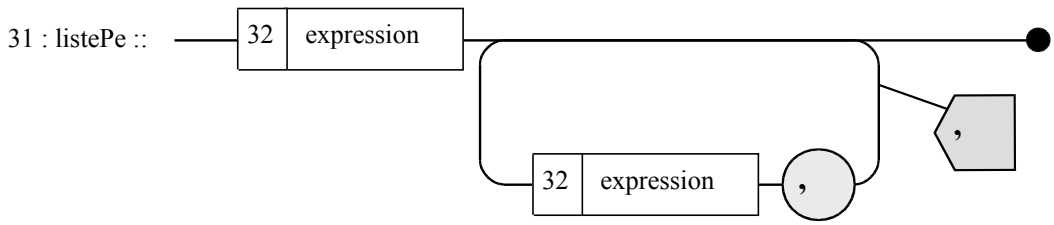

20 : corpsFonct::
25 partieDeclaProc
begin
29 suiteInstrNonVide
end
begin
21 : partieFormelle ::
22 listeSpecifFormelle
(
)
)
22 : listeSpecifFormelles ::
23 specif
;
23 specif
23 : specif ::
in
24 mode
28 listeIdent
:
identificateur
out
out
24 : mode ::
in
25 : partieDeclaProc ::
26 listeDeclaVar
26 : listeDeclaVar ::
27 declaVar
identificateur
27 declaVar

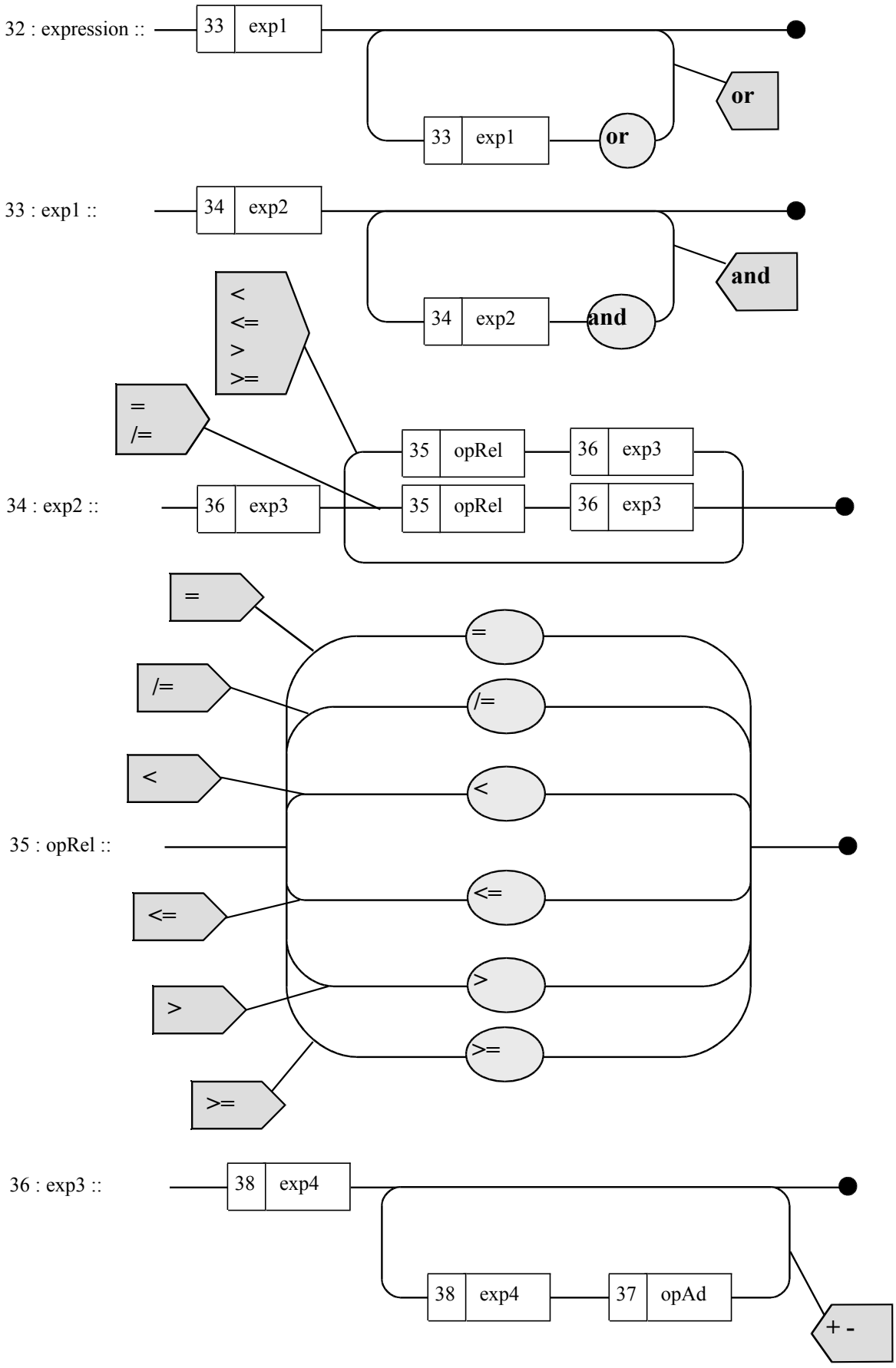

declaVar ::
28
listeIdent
:
identificateur
;

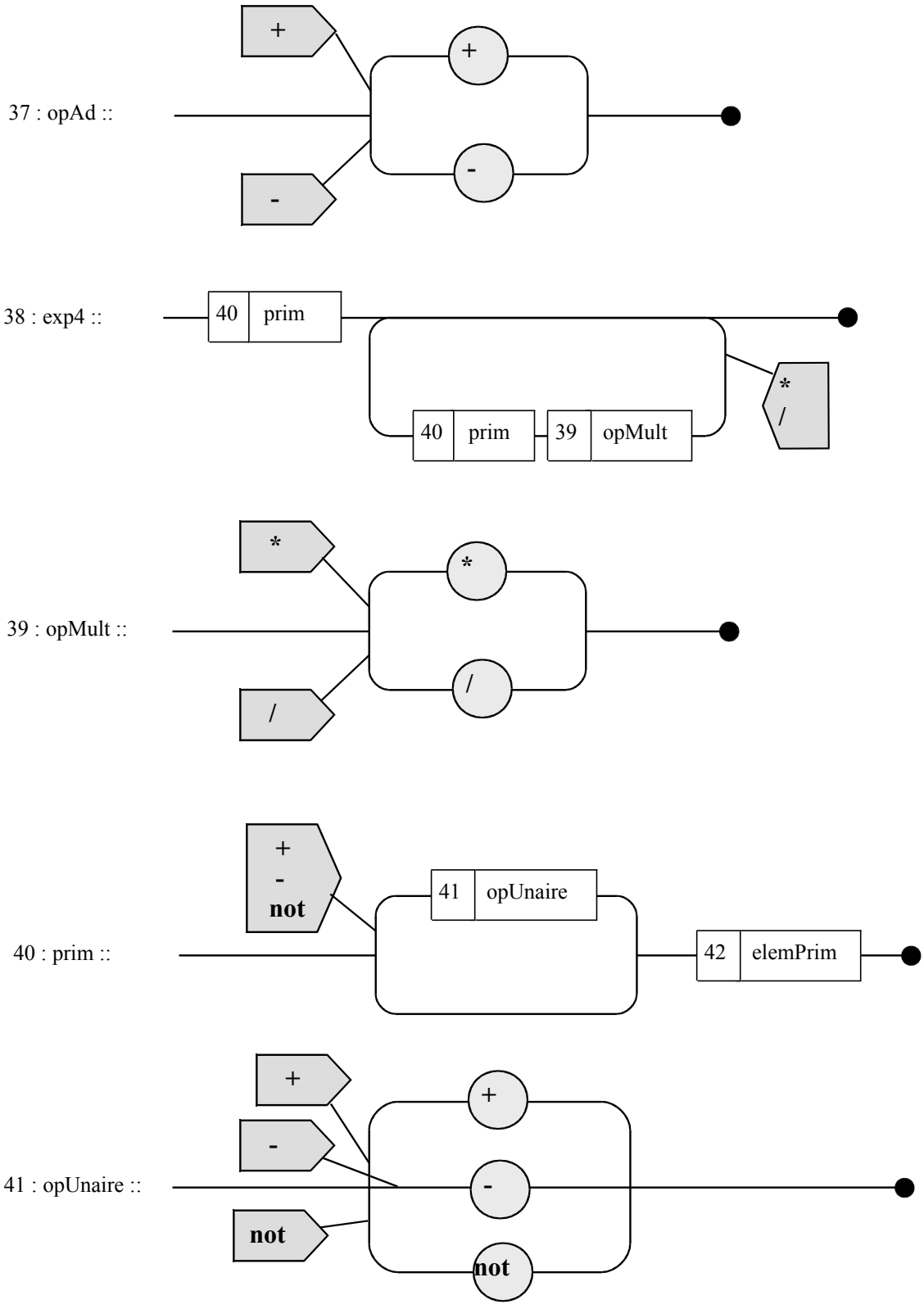

28 : listeIdent ::
identificateur
,
identificateur
,

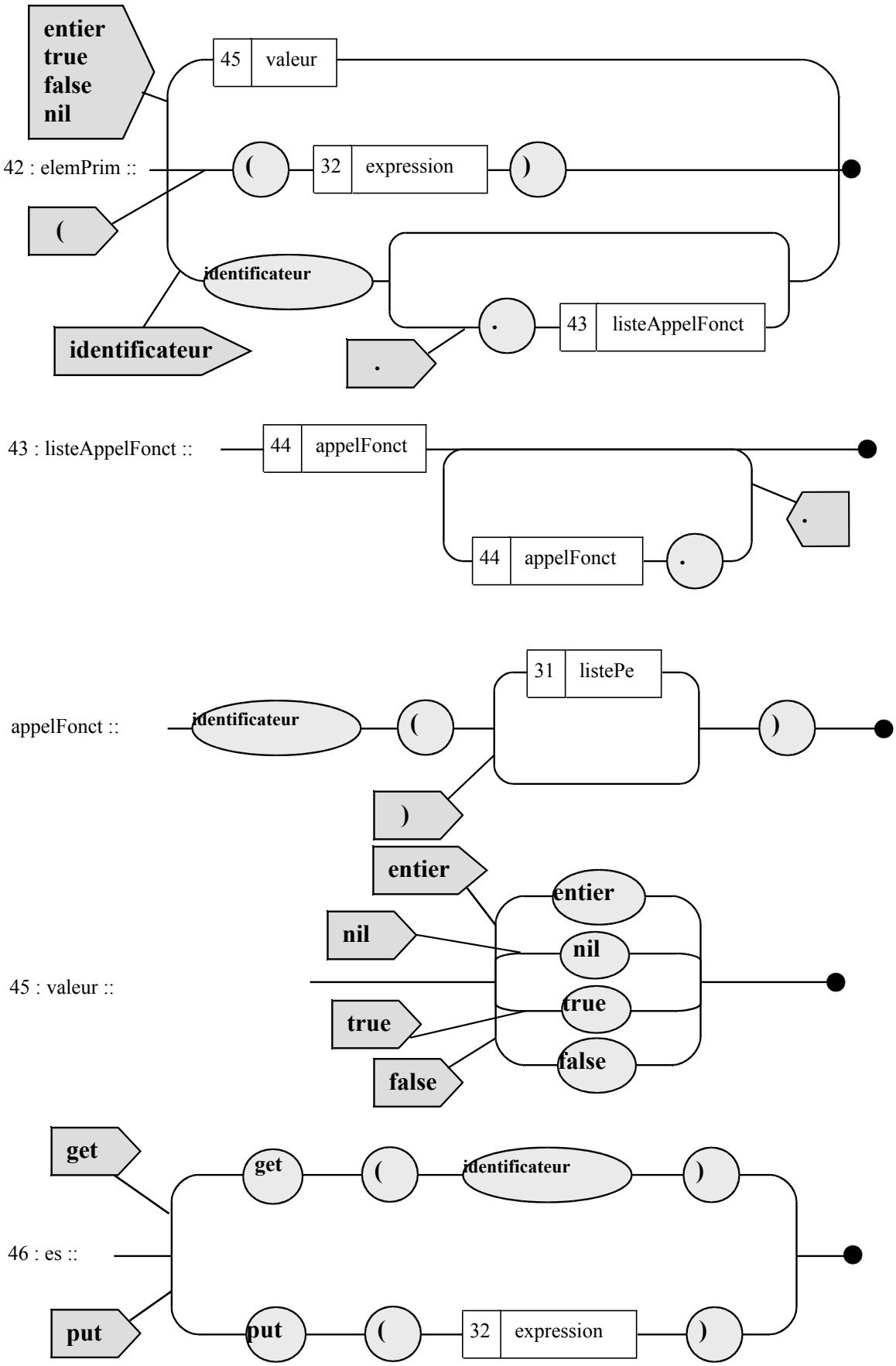

29 : suiteInstrNonVide ::
30
instr
;
30
instr
;


identificateur
identificateur
.
.
43
listeAppelFonct
:=
:=
32
expression
while
47
boucle
if
48
altern
30 : instr ::
get
put
46
es
return
49
retour
error
(
32
expression
)


31 : listePe ::
32
expression
32
expression
,
,


32 : expression ::
33
exp1
or
33
exp1
or
33 : exp1 ::
34
exp2
and
34
exp2
and
<
<=
>
>=
=
/=
34 : exp2 ::
36
exp3
35
opRel
36
exp3
35
opRel
36
exp3
=
/=
<
35 : opRel ::
<=
>
>=
=
/=
<
<=
>
>=
36 : exp3 ::
38
exp4
38
exp4
37
opAd
+ -


37 : opAd ::
+
-
+
-
38 : exp4 ::
40
prim
40
prim
39
opMult
*
/
39 : opMult ::
*
/
*
/
40 : prim ::
+
-
not
41
opUnaire
42
elemPrim
41 : opUnaire ::
+
-
not
+
-
not


entier
true
false
nil
45
valeur
42 : elemPrim ::
(
32
expression
)
(
identificateur
identificateur
.
43
listeAppelFonct
.
43 : listeAppelFonct ::
44
appelFonct
44
appelFonct
.
.
appelFonct ::
identificateur
(
31
listePe
)
)
entier
nil
true
false
45 : valeur ::
entier
nil
true
false
get
put
46 : es ::
get
(
identificateur
)
put
(
32
expression
)

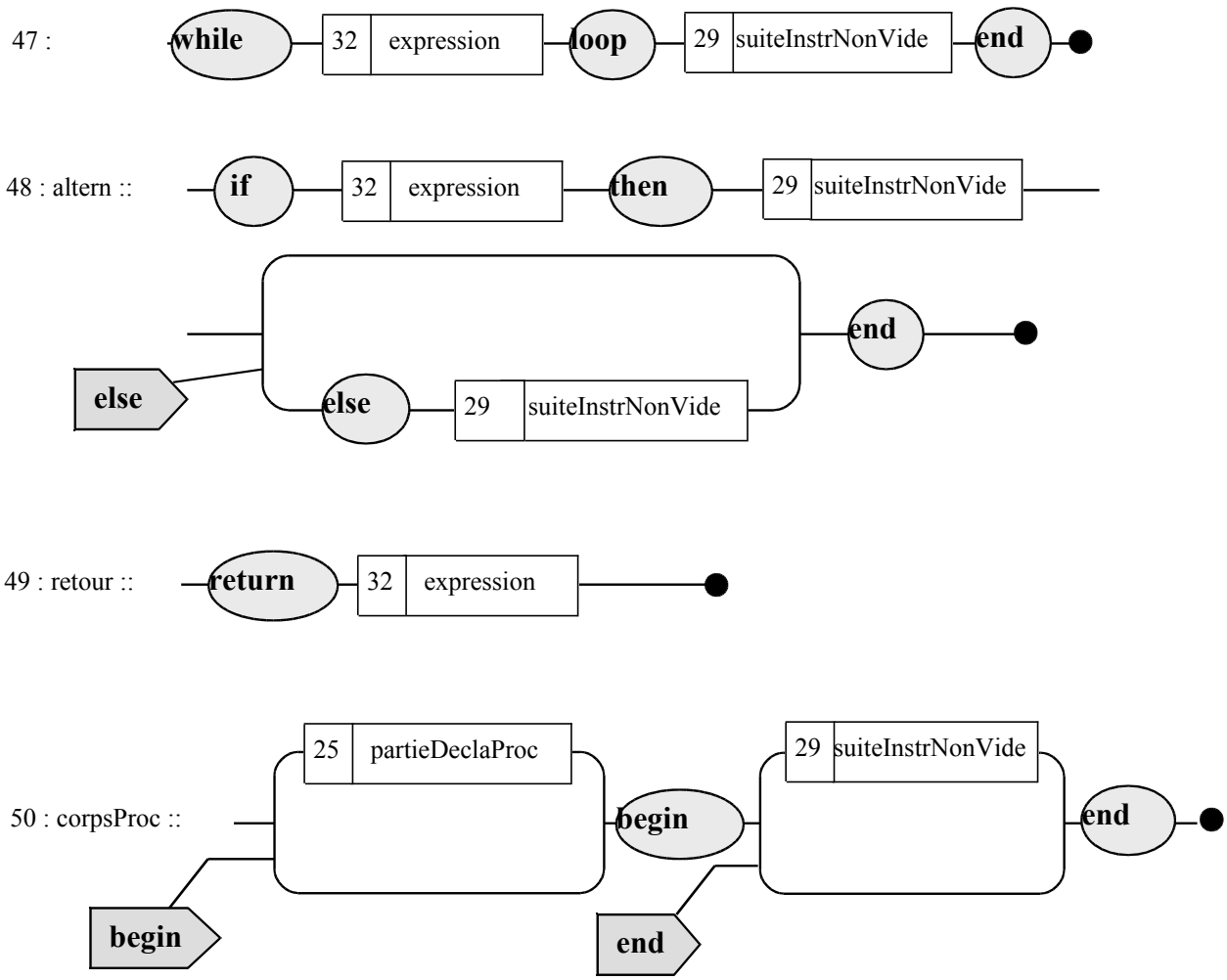


Figure 4. Diagrammes syntaxiques nilNovi Objet

6. Interpréteur

A priori il ne vous sera pas nécessaire de modifier les programmes source de l'interpréteur. Pour cette raison, nous ne fournissons ici que l'essentiel nécessaire à son utilisation.

L'architecture logicielle employée est l'image de la machine nilNovi Objet telle qu'elle est présentée dans [1]. Le programme objet est un tableau (de longueur statique) d'instructions. Chaque instruction nilNovi Objet est un objet contenant 1, 2 ou 3 champs selon la structure de l'instruction.

Les méthodes principales de la classe *interpreteur* (fichier *src/interpreteurNNO/programmeObjetNNO/interpreteur.java*) sont les suivantes :

- *interpreteur* : constructeur, doté de deux paramètres formels zoneTA2 et zoneTA3 qui sont des éditeurs de texte. Ces deux paramètres devront être associés aux paramètres effectifs jTextArea2 et jTextArea3 qui sont respectivement les zones de l'interface fenêtrée où est produit le code désassemblé et où s'effectue l'affichage des sorties

utilisateur (voir la méthode *jButton4_actionPerformed* de la classe *Cadre1* du fichier *src/nno/Cadre1.java* l. 339 pour un exemple de création d'un interpréteur).

- *interpreter* : méthode sans paramètre qui interprète le code objet (voir la méthode *jButton5_actionPerformed* de la classe *Cadre1* du fichier *src/nno/Cadre1.java* l.482 pour un exemple d'appel de l'interpréteur).

- *desassembler* : méthode sans paramètre qui désassemble le code objet existant (voir *jButton7_actionPerformed* de la classe *Cadre1* du fichier *src/nno/Cadre1.java* l. 514 pour un exemple d'appel du désassembleur).

- *debutProg*, …, *traVirt* : méthodes dont l'invocation produit une nouvelle instruction dans le programme objet, à la suite des instructions existantes. Voir la méthode *genCode* dans la classe *PointsGeneration* du fichier *src/PtGen/PointsGeneration.java* pour des exemples d'utilisation de ces méthodes.

- *getValCE : (obtenir la valeur du compteur d'emplacement)*. Le compteur d'emplacement est le compteur utilisé pour *créer* le programme objet. Il désigne l'emplacement de la dernière instruction créée. Cette méthode est nécessaire pour la compilation car lorsque l'on crée une instruction incomplète (ceci peut survenir car certaines informations ne sont pas disponibles au moment de la création de l'instruction), il faut pouvoir revenir ultérieurement sur l'instruction pour la compléter. Il faut donc pouvoir connaître son adresse dans le programme objet. C'est ce que permet cette méthode.

- *ModifInstr :* cette méthode complète la méthode précédente (*getValCE*) en ce qu'elle permet de modifier le premier argument d'une instruction donnée du programme objet.

7. La TDI

La construction de la TDI se fait par un algorithme dont le rôle est d'intégrer une par une les entités rencontrées (les classes et les variables globales) ainsi que leurs différentes constituants (les attributs, les méthodes, … pour les classes). Cependant le choix de la démarche de compilation (dirigée par la syntaxe) fait qu'il n'est pas possible d'organiser cette construction à la manière d'un algorithme classique[5], il faut au contraire « éclater » l'algorithme en petits fragments qui seront appelés à des endroits adéquats soit directement dans l'analyse syntaxique soit indirectement par l'intermédiaire des points de génération.

Une fois la TDI construite, mais aussi pendant la construction, il faut exploiter la représentation pour tenter d'y retrouver des informations (comme par exemple l'existence d'un identificateur, ou les adresses) nécessaires à la détection d'erreurs « sémantiques » ou à la compilation proprement dite. Ceci nous conduit à mettre en évidence, outre les méthodes de construction de la TDI, une seconde classe de points d'accès à la TDI, les « observateurs ».

7.1 Navigation et adressage dans la TDI

Tout élément présent dans la TDI peut être adressé soit de manière symbolique (par son identificateur) soit de manière absolue (par son adresse statique). Ainsi, dans l'exemple de la procédure *aeronef* du polycopié de TD (p. 21), la classe « avion » est connue soit par cet identificateur, soit par son adresse statique (2 par exemple).

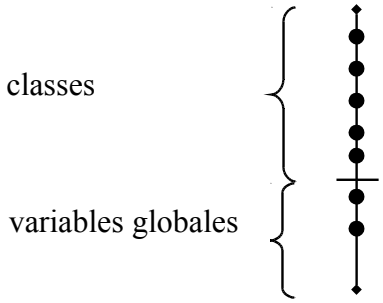
La configuration de la TDI est faite pour permettre un accès direct à partir de l'adresse statique, alors que l'identificateur symbolique exige une recherche (séquentielle en

[5] Cela serait cependant possible en utilisant la technique des coroutines permises par des langages comme Simula.

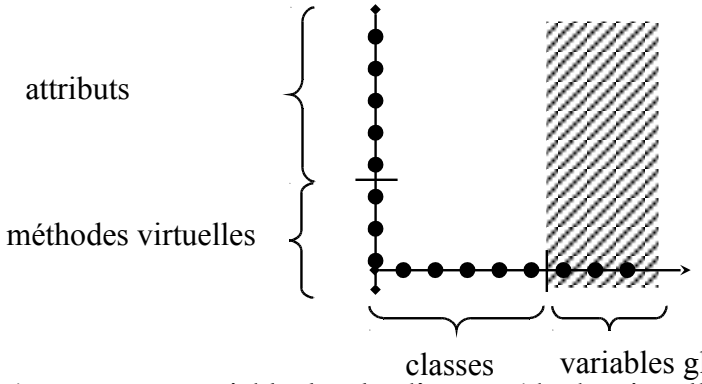
l'occurrence). C'est pourquoi nous conseillons de favoriser l'adresse statique. à chaque fois que c'est possible.

Par ailleurs la dimension de l'espace d'adressage dépend de la nature de l'élément adressé :

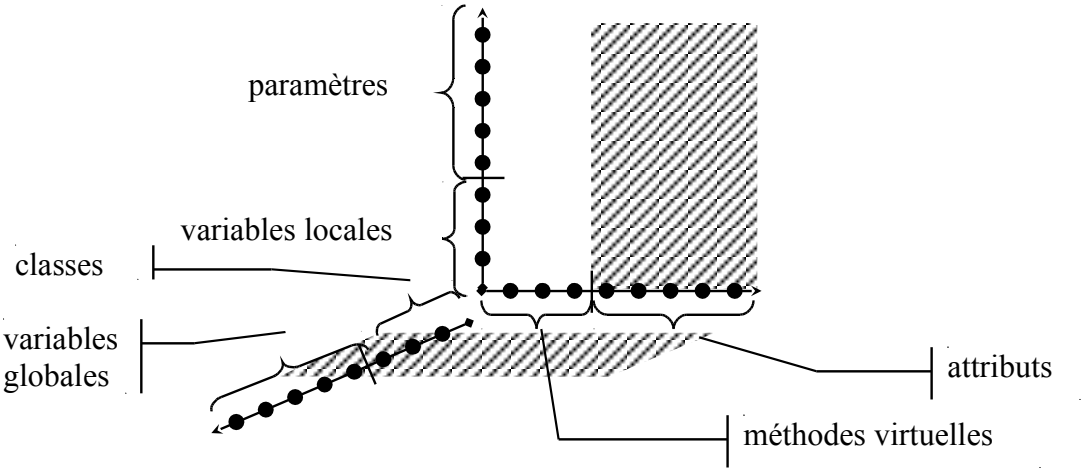
- Une classe ou une variable globale se repère dans un espace à une dimension. Ses coordonnées sont des singletons.



- Un attribut ou une méthode virtuelle dépend d'une classe. Il se repère dans un espace à deux dimensions. Ses coordonnées (symboliques ou absolues) sont des couples (adresse de la classe, adresse de l'attribut) ou (adresse de la classe, adresse de la méthode virtuelle). Il en est de même des paramètres et des variables locales du constructeur d'une classe.



- Un paramètre ou une variable locale d'une méthode virtuelle dépend de la méthode virtuelle en question et de la classe. Ils se repèrent dans un espace à 3 dimensions.



Ainsi en général quand une méthode de la TDI fait référence à une entité donnée, le nombre de paramètres utilisés est égal à la dimension de l'espace dans lequel il se trouve. Par exemple pour recherche le type du paramètre d'une méthode virtuelle, il faudra fournir l'« adresse » des 3 constituants classe/méthode/paramètre. Il existe cependant des exceptions, quand une ou plusieurs des coordonnées sont implicites. Ainsi par exemple si on veut créer un attribut, il est inutile de mentionner la classe considérée, c'est la classe *courante*, celle qui est *en cours* de construction.

7.2 Identificateur de classe

Lorsque l'on rencontre un identificateur de classe, on peut créer une entrée de type « classe » dans la TDI. Cette entrée est pour l'instant incomplète puisqu'elle ne contient que cet identificateur et son adresse statique. Par ailleurs c'est aussi à ce moment qu'il faut tenter de reconnaitre un identificateur de classe doublement déclaré afin de signaler l'erreur au programmeur NNO.

7.3 Classe fille

Dans le cas d'une classe fille, à la rencontre de l'identificateur de la mère, on doit (si la classe mère existe, le cas échéant étant un cas d'erreur) faire hériter à la classe fille :

- les attributs de la classe mère (avec les informations associées : type et adresse statique),

- les méthodes virtuelles, qui, quel que soient leurs statuts dans la classe mère, deviennent des méthodes « héritées ».

Techniquement cet héritage ne peut être simplement un partage d'information entre mère et fille. En effet, la liste des attributs hérités va être enrichie par l'adjonction des attributs propres. Ceci ne doit pas avoir de conséquences sur la classe mère. De même, du côté méthodes virtuelles, comme nous venons de le voir, l'héritage est partiel, puisque les méthodes filles ont toutes le statut de méthodes héritées. En conséquence il faut procéder par « clonage » et non par partage.

7.4 Attributs

Nous ne revenons pas sur les attributs hérités, leur traitement n'est pas remis en cause. Chaque attribut propre rencontré doit être introduit dans la liste d'attributs avec son identificateur, son type et son adresse statique. Il faut au préalable s'assurer qu'aucun attribut de même nom n'existe dans la classe considérée.

Une difficulté doit cependant être correctement reconnue et traitée. C'est celle dite de la « reprise ». Compte tenu des possibilités syntaxiques offertes au programmeur NILNOVI OBJET pour déclarer ses attributs, comme dans :

```
a, b, i, k : integer ;
```

la distance entre la reconnaissance d'un identificateur d'attribut et entre son type peut être importante, et par ailleurs, lors de la rencontre du type, il faut « reprendre » tous les descripteurs d'attributs incomplets afin de leur ajouter la nouvelle information.

Cette difficulté se retrouve à d'autres occasions : liste de paramètres formels, de variables locales ou globales.

7.5 Déclaration de méthode

Pour chaque méthode virtuelle rencontrée dans l'interface NILNOVI OBJET, il faut élaborer une représentation temporaire de la méthode (méthodes *creerFonct* et *creerProc* de l'interface Java *PListeEnt*). À l'issue du traitement de l'entrée dans l'interface et si la méthode considérée est une méthode propre à la classe en cours de compilation, une entrée

définitive est créée dans la TDI (méthodes *creerFonctPropre* et *creerProcPropre* de l'interface Java *PListeEnt*).

Le passage par une représentation intermédiaire n'est pas nécessaire pour les constructeurs.

Le traitement de la liste des paramètres va exiger d'employer la technique de reprise présentée au §7.4

7.6 Définition de méthode

Lors du traitement de la définition d'une méthode, on doit vérifier que son en-tête est compatible avec la déclaration. Les identificateurs de variables locales sont traitées par une technique de reprise (cf. §7.4). À l'issue de la définition de la méthode, les identificateurs de variables locales doivent être explicitement supprimés de la TDI (en utilisant les méthodes *supVarLocXX* de l'interface Java *PListeEnt*).

A la fin du traitement d'une méthode-fonction, on doit s'assure qu'il y a bien au moins une instruction *return* dans la partie impérative de la méthode.

7.7 Parties impératives

La TDI est uniquement *consultée* dans les parties impératives (que ce soit dans les méthodes ou dans le programme principal), afin de vérifier que les expressions et les instructions sont sémantiquement correctes.

8. Utilisation des analyseurs, points de génération, exemple

L'analyseur syntaxique (fichier *src/analyseurSyntaxique/analyseurLL1.java*) contient *aL*, instance de l'analyseur lexical. *aL* (l. 35) est une liste (linkedList en Java) d'unités lexicales. L'unité lexicale courante est accessible par la méthode *getUl*(). Cette unité lexicale est de l'un des 5 types autorisés (caractère, entier, identificateur, mot clé ou unité de fin de liste). Dans le cas d'un caractère (resp. d'un entier, identificateur ou mot clé) on utilise la méthode *cC*() (resp. *val*(), *ch*() ou *mC*()) pour accéder au champ contenant l'unité lexicale dans l'appel d'un point de génération (Exemple : méthode *declaClasse*, l. 119).

Ainsi que nous l'avons mentionné ci-dessus, les programmes java fournis contiennent – à titre d'exemples – quelques points de génération qui, appelés par l'analyseur syntaxique, vont permettre :

- de créer un programme objet,
- d'ébaucher l'élaboration de la TDI en y introduisant les identificateurs de classe et les identificateurs de variables globales,
- d'afficher la partie de la TDI créée.

Ces points de génération montrent par la même occasion comment utiliser « l'interface » *PListeEnt* (qui permet de gérer la TDI sans en connaitre la structure intime)

Le fichier *src/PtGen/PointsGeneration.java* décrit la classe *PointsGeneration*. Celle-ci contient en particuliers les attributs :

- *l*, de type *PListeEnt*. C'est la (référence à) la TDI.

- *varGlo* de type *boolean*. Cet attribut permet de savoir si on analyse ou non un identificateur de variable globale. Dans le premier cas on l'introduit dans la TDI dans le second on l'ignore.

La classe *PointsGeneration* contient les méthodes :

- *PointsGeneration*, constructeur de la classe.

- *debut,* méthode qui crée notamment une instance de la TDI (l. 40).

- *creerClasse,* méthode qui introduit un identificateur de classe dans la TDI (à condition que cet identificateur n'existe pas déjà et ne soit pas un « mot clé »). L'insertion se fait par appel à la méthode *creerClasse* de l'interface java *PListeEnt* (l. 51). *creerClasse* est l'une des méthodes documentées dans *src/documentation/PListeEnt.html*.

- *flagVarGloOn,* méthode qui permet d'affecter à *true* l'attribut booléen *varGlo*.

- *flagVarGloOf,* méthode qui permet d'affecter à *false* l'attribut booléen *varGlo*.

- *insererIdentGlo*, méthode qui place son paramètre *identVarGlo* dans la TDI, à condition qu'il s'agisse d'un identificateur de variable globale. On note que cette méthode fait appel à la méthode *creerVarGlo* de l'interface java *PListeEnt* (l. 65). *creerVarGlo* est l'une des méthodes documentées dans *src/documentation/PListeEnt.html*.

- *fin,* méthode qui affiche l'état de la TDI.

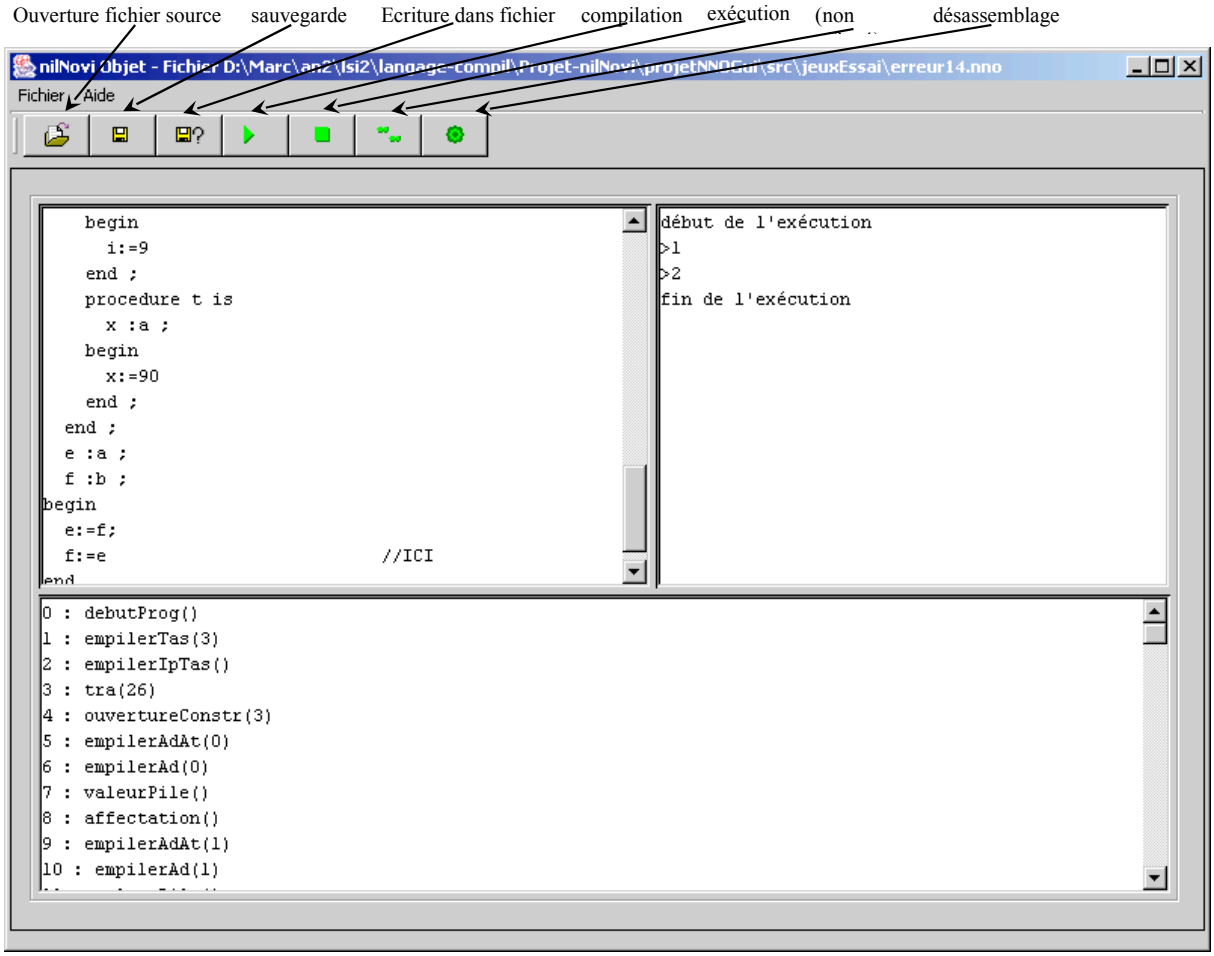
- *genCode,* méthode qui crée un programme objet (sans relation avec le programme nilNovi Objet traité) afin d'illustrer l'action des boutons « exécuter » et « désassembler » de l'interface fenêtrée nilNovi Objet.

Ces points de génération sont utilisés dans l'analyseur syntaxique (fichier *src/analyseurSyntaxique/analyseurLL1.java*). Le diagramme syntaxique où est fait l'appel est mentionné en guise de commentaire dans l'entête du point de génération. L'appel est signalé par un

```
//PG
```

dans le code de l'analyseur.

9. Interface graphique



Annexe 2
Grammaires et traitement des priorités

Cette annexe peut être sautée en première lecture. Elle décrit comment nous sommes passés de la grammaire fournie dans [1] aux diagrammes syntaxiques de l'annexe 1. Considérons un langage des expressions arithmétiques (petit sous-ensemble du NILNOVI OBJET) comprenant les 4 opérateurs binaires usuels et dans lequel, pour simplifier, les opérandes se réduisent à un chiffre :

```
(9+(5*3-1)-5)+3
```

Grammaire minimale mais ambiguë

La grammaire G1 suivante (axiome <exp>) décrit la syntaxe de ce langage :

```
<exp> ::=          <exp> <op> <exp> |
                   ( <exp> ) |
                   <chiffre>
<op> ::=           +|-|*|/
<chiffre> ::=      0|…|9
```

Cependant elle présente deux défauts majeurs pour la réalisation d'un compilateur d'expressions arithmétiques :

1. Elle est ambiguë : une expression telle que
   ```
   5-1-2
   ```
   peut être analysée comme ((5-1)-2) ou comme (5-(2-1)).
2. Elle ne rend pas compte de la plus forte priorité des opérateurs multiplicatifs sur les opérateurs additifs. Une des interprétations possibles pour l'analyse de
   ```
   5-2*3
   ```
   est
   ```
   (5-2)*3
   ```
   ce qui est contraire à la convention habituelle pour la priorité de l'opérateur – par rapport à l'opérateur *.

Peut-on rédiger une grammaire qui n'aurait pas ces inconvénients ? La réponse est oui comme nous allons le voir dans la section suivante.

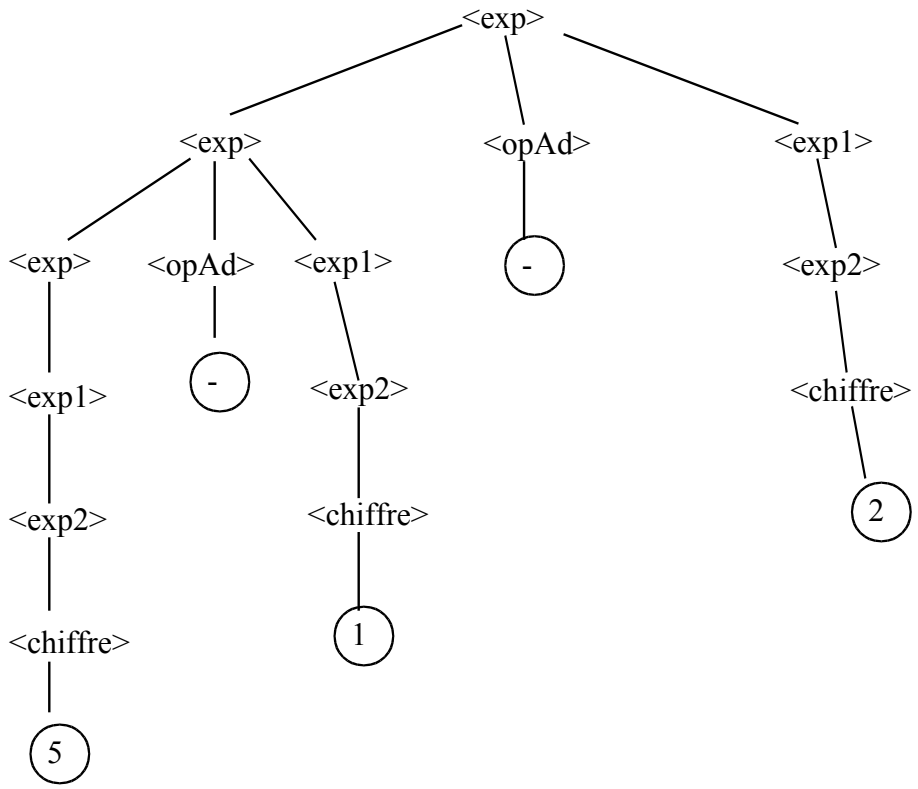
Une grammaire qui gère correctement les priorités

La grammaire G2 suivante (axiome <exp>) décrit le même langage que la grammaire G1. En outre, comme nous allons le voir, elle rend compte naturellement des règles de priorité habituelles :

```
<exp> ::=          <exp> <opAd> <exp1> | <exp1>
<exp1> ::=         <exp1> <opMult> <exp2> | <exp2>
<exp2> ::=         ( <exp> ) | <chiffre>
<opAd> ::=         +|-|
<opMult> ::=       *|/
<chiffre> ::=      0|...|9
```

Considérons l'analyse de l'expression

```
5-1-2                                   (E1)
```



Cette analyse produit la forme postfixée 51-2- qui correspond à l'expression complètement parenthèsée ((5-1)-2). C'est le résultat attendu.

Par contre cette grammaire G2 présente une caractéristique qui, dans le contexte de l'écriture d'un analyseur descendant gauche-droite, est un défaut rédhibitoire : elle est récursive à gauche !

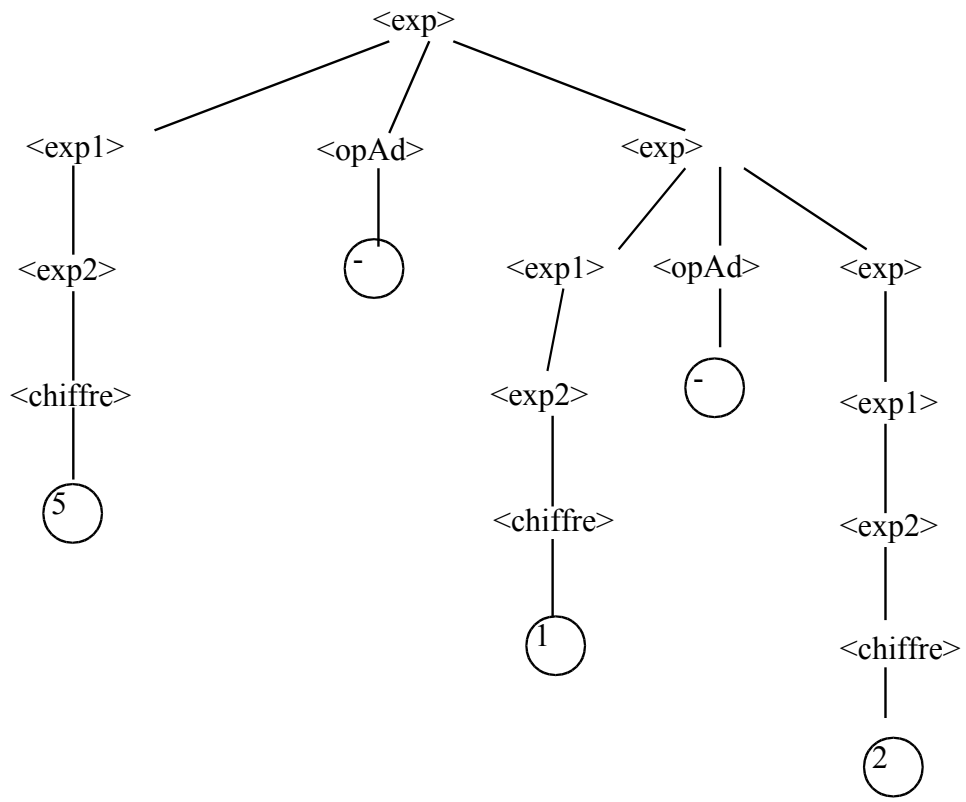
On peut chercher à supprimer ce défaut. C'est ce qui est fait dans la section suivante.

Une grammaire sans récursivité gauche

La grammaire G3 suivante (axiome <exp>) décrit le même langage que les grammaires G1 et G2, elle n'est pas récursive à gauche.

```
<exp> ::=        <exp1> <opAd> <exp> | <exp1>
<exp1> ::=       <exp2> <opMult> <exp1> | <exp2>
<exp2> ::=       ( <exp> ) | <chiffre>
<opAd> ::=       +|-|
<opMult> ::=     *|/
<chiffre> ::=    0|...|9
```

La priorité des opérateurs multiplicatifs sur les opérateurs additifs est correctement représentée par cette grammaire. Par contre nous avons fait un pas dans la mauvaise direction puisque pour un opérateur donné, cette grammaire modélise une priorité de droite à gauche. En effet, considérons à nouveau l'expression (E1) ci-dessus :



Cette analyse produit la forme postfixée 512- -qui correspond au parenthésage (5-(1-2)). Ce résultat n'est pas celui qui est attendu !

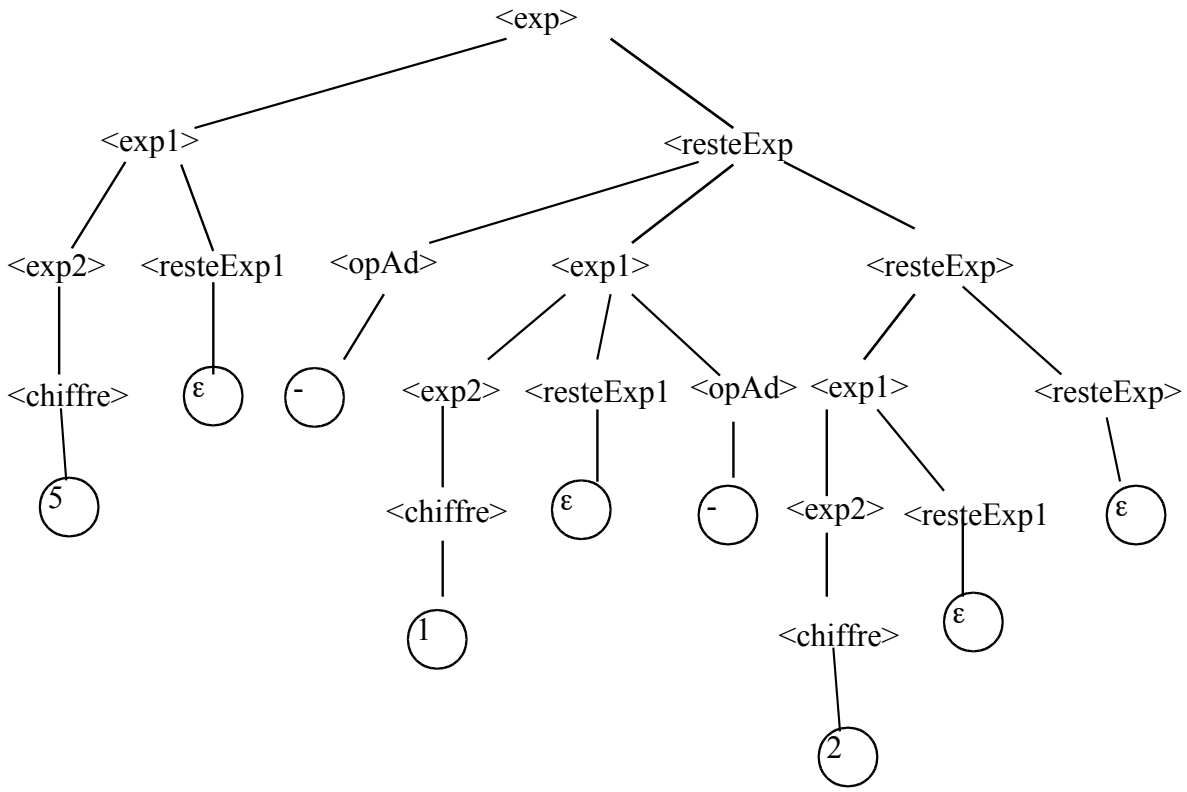
Comment concilier tous les avantages de G2 et de G3 tout en faisant disparaître les inconvénients de G3 ?

Une grammaire sans récursivité gauche qui gère correctement la priorité des opérateurs

La grammaire G4 suivante (axiome <exp>) décrit le même langage que les grammaires G1, G2 et G3. Elle gère correctement les règles de priorités usuelles.

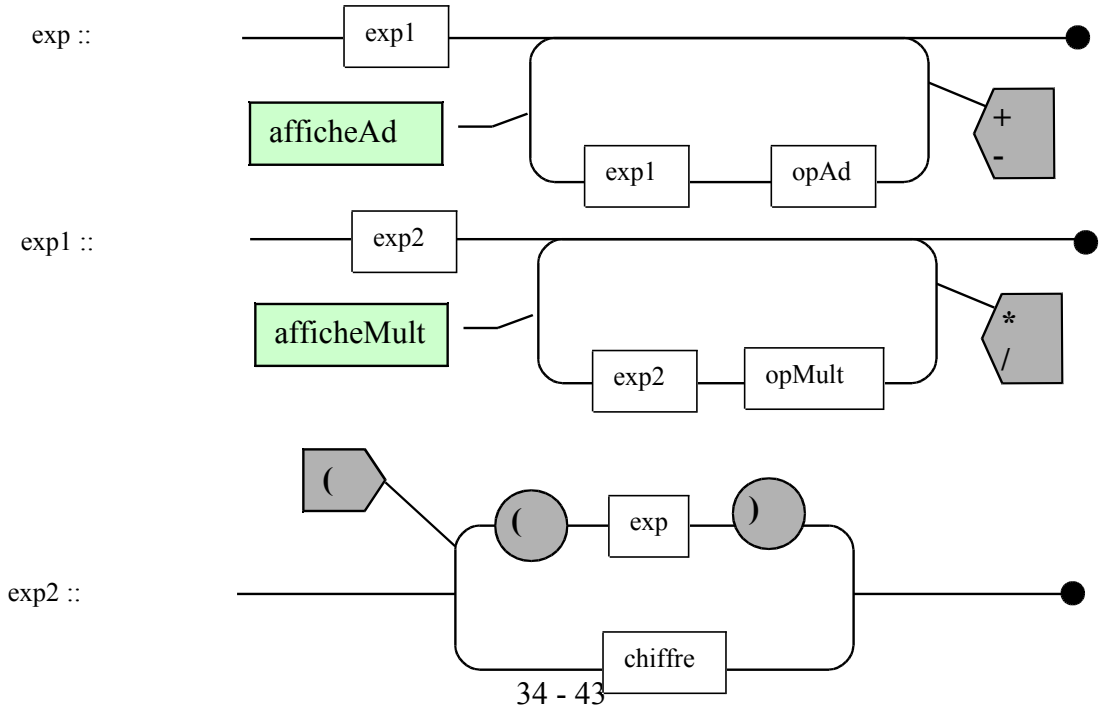
```
<exp> ::=         <exp1> <resteExp1>
<resteExp1> ::=   <opAd> <exp1> <resteExp1> | ε
<exp1> ::=        <exp2> <resteExp2>
<resteExp2> ::=   <opMult> <exp2> <resteExp2> | ε
<exp2> ::=        ( <exp> ) | <chiffre>
<opAd> ::=        +|-|
<opMult> ::=      *|/
<chiffre> ::      0|...|9
```

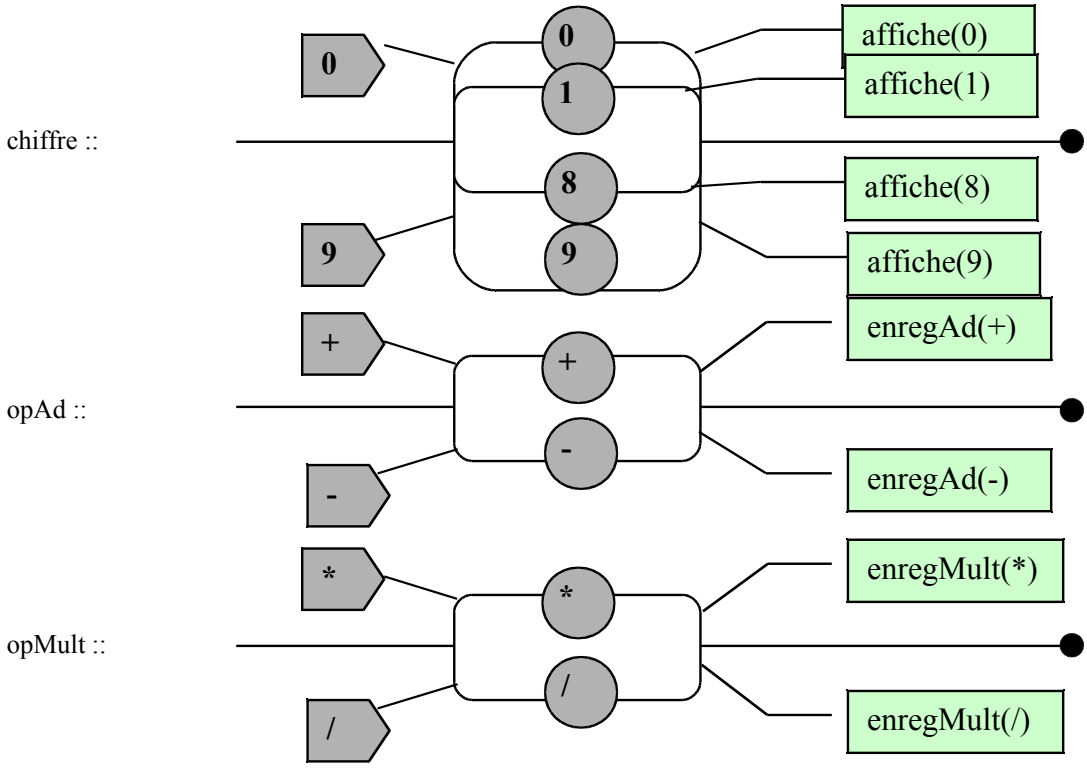
On analyse à nouveau l'expression (E1).



Cette analyse produit la forme postfixée 51-2- qui correspond à l'expression complètement parenthèsée ((5-1)-2). C'est le résultat attendu.

Il est facile de transformer cette grammaire G4 en une collection de diagrammes syntaxiques. On peut en profiter pour éliminer la récursivité droite en la remplaçant par des « boucles » et pour placer des panneaux d'aiguillage qui faciliteront le codage. C'est cette technique qui a été utilisée pour la transformation de la grammaire de NILNOVI OBJET dans les diagrammes syntaxiques de l'annexe 1.





L'affichage infixé de l'expression peut s'obtenir en plaçant des points de génération comme dans les diagrammes ci-dessus :

- On dispose de deux « registres » (le registre additif et le registre multiplicatif) capables de mémoriser respectivement un opérateur additif et un opérateur multiplicatif.
- La procédure *enregAd* (resp. *enregMult*) enregistre son paramètre dans le registre additif (resp. multiplicatif).
- La procédure *affiche*(*i*) affiche la valeur *i*.
- La procédure *afficheAd* (resp. *afficheMult*) affiche l'opérateur situé dans le registre additif (resp. multiplicatif).

Il est facile de voir que l'analyseur enrichi des points de génération va permettre d'afficher les expressions analysées sous la forme postfixée.

Annexe 3
Proposition pour l'organisation du travail pendant le projet

Déroulement du projet

1. Lecture du sujet (1h).
2. Prise en main des outils et de la fourniture (2h).
3. Répartition du travail entre les membres du groupe (1/2h).
4. Conception (12h30)
5. Développement, intégration (9h)
6. Tests et corrections ultimes (remplissage des fiches de test ci-dessous) (2h)
7. Documentation (3h)

Conception et développement

On peut mettre en évidence les tâches suivantes :

1. identifier les diagrammes syntaxiques qui doivent délivrer un résultat afin d'effecteur un contrôle sémantique du texte source (cf. polycopié de TD),
2. aménager l'analyseur dans le sens du point 1,
3. identifier, spécifier et placer les points de génération sur les diagrammes syntaxiques,
4. coder les points de génération,
5. insérer les appels des points de génération dans l'analyseur, conformément au point 3.

Références

1. M. Guyomard. Compilation NILNOVI – support de cours. Année 2008-2009. Enssat. Université de Rennes I.
2. Sites web :

— http://www.math.grin.edu/~stone/courses/software-design/lex.html

— http://www.iosphere.net/~ian/personal/ddj2.htm

— http://www.ecst.csuchico.edu/~bhsteel/250/examplesHandout/handout.html

— http://www.uafcs.alaska.edu/~cs631/node15.html

— http://www.csc.calpoly.edu/~gfisher/450/doc/yacc/paper.txt

— http://www.ee.ic.ac.uk/course/advanced/yacc/yacc.html

— http://www.cti.ecp.fr/~bernare8/textes/minimanlexyacc.html

— http://www-isia.cma.fr/~bernard/textes/minimanlexyacc.html

— http://www.math.grin.edu/~stone/courses/software-design/yacc.html

— http://www.via.ecp.fr/~eb/textes/minimanlexyacc.html

— http://vcapp.csee.usf.edu/~sundares/lex_yacc.html

— http://members.xoom.com/thomasn/y_man.htm

— http://www.enc.hull.ac.uk/people/pjp/Teaching/08208-9697/08208IntroductoryLab.html

— http://www.dscpl.com.au/ose-4.3/mk_22.htm

Table des matières

NNO (répertoire jeuEssaiNNO) – jeux d'essai avec erreur à la compilation

Noms : …… mettre une X dans la colonne X si test correct

n° programme	Objectif du test	X	Remarques
error1	Redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, nombre de paramètres différents		
error2	Redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, type de paramètres différents		
error3	Redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, mode de passage différent		
error4	Redéfinition d'une méthode-procédure par une méthode-procédure avec un profil différent, nom de paramètres différents		
error5	Redéfinition d'une méthode-fonction par une méthode-procédure		
error6	*this* déclaré comme identificateur de paramètre formel		
error7	*this* utilisé comme identificateur d'attribut		
error8	*this* utilisé comme identificateur de classe		
error9	Redéfinition d'un attribut dans une même hiérarchie		
error10	*integer* comme identificateur d'attribut		
error11	*integer* comme identificateur de classe		
error12	Classe sans constructeur		
error13	Double déclaration de paramètre formel dans une méthode		
error14	Identificateur d'attribut doublement déclaré		
error15	Double déclaration d'un identificateur de classe		
error16	Redéfinition d'un paramètre par une variable locale d'une méthode		
error17	Redéfinition d'une opération dans la même classe		
error18	Conflit attribut/classe		
error19	Appel d'une méthode incompatible avec sa définition		
error20	Affectation *this* dans une méthode		
error21	Appel d'un constructeur incompatible avec sa déclaration : appelé avec une classe différente de celle ou il est défini		
error22	Appel d'un constructeur incompatible avec sa déclaration : appelé avec un objet (et non une classe)		

error23	La définition d'une méthode est différente de sa déclaration : fonction devient procédure		
error24	Type délivré par une fonction différent du type déclaré		
error25	Problème de *return*		
error26	Appel d'une méthode spécifique au type dynamique		
error27	Paramètre formel de mode *in out* : paramètre effectif de mode *in*		
error28	Une méthode déclarée n'est pas définie dans la classe		
error29	Fonction avec paramètre de mode *in out*		
error30	Constructeur avec paramètre de mode *in out*		
error31	*return* dans un constructeur		
error32	*return* dans la partie impérative		
error33	Héritage depuis une classe qui n'existe pas		
error34	Plusieurs constructeurs		
error35	Pas de constructeur		
error36	*return* dans une méthode-procédure		
error37	Accès à attribut hors classe		
error38	*this* utilisé en dehors d'une classe		
error39	Type délivré effectivement supérieur au type déclaré		
error40	Double déclaration de méthode-procédure		
error41	Affectation de variables objet incompatibles		
error42	Un constructeur appelé comme une procédure		
error43	Fonction qui modifie un paramètre formel		
error44	Pas de *return* dans le corps d'une fonction		
error45	Classe qui hérite d'elle même		

NNO (répertoire jeuEssaiNNO) – jeux d'essai sans erreur à la compilation ni exécution

Noms : …… mettre une X dans la colonne X si test correct

n° programme	Objectif du test	X	Remarques
correct1	Redéfinition correcte d'un identificateur d'attribut par un identificateur de constructeur		
correct2	Redéfinition correcte d'un identificateur d'attribut par un identificateur de variable locale		
correct3	Test de "error"		
correct4	Essai de polymorphisme et de liaison dynamique		
correct5	Manipulation de listes d'objets		
correct6	Imbrication d'appels de méthodes-fonctions		
correct7	Méthodes identifiées par integer et par nil		
correct8	Essai pronom *this* et passage de paramètre		
correct6	Définition de méthodes dans un ordre différent de leur déclaration		
correct10	Évaluation d'expressions arithmétiques en NNO		
correct11	Expression d'objet complexe : appel d'une fonction qui délivre un objet		
correct12	Appel récursif d'une méthode-procédure et pronom « *this* »		

NNA (répertoire jeuEssaiNNA)

Noms : …… mettre une X dans la colonne X si test correct

n° programme	**Objectif du test**	**X**	**Remarques**
	Programmes corrects		
correct1	Boucles imbriquées		
correct2	Expression relationnelle avec des booléens		
correct3	Evaluation de conditions complexes		
correct4	Affiche le nombre de valeurs paires lues		
	Programmes incorrects		
error1	Lecture d'un booléen		
error2	Ecriture d'expressions booléennes		
error3	Affectation : typage incorrect		
error4	Condition incorrecte dans une alternative		
error5	Condition incorrecte dans un *while*		
error6	Double déclaration de variable		