- Génération d'analyseur lexical et syntaxique - NF11

11 juin 2010

Table des matières

Table des figures

Introduction

Au cours de l'histoire de l'humanité, différentes formes d'écriture et de langages se sont développées afin de permettre aux hommes de communiquer, de partager et d'échanger des informations. Aujourd'hui, un programme codé dans un langage informatique est même reconnu comme étant une nouvelle forme d'écriture et d'expression de la personnalité du développeur avec la reconnaissance du droit d'auteur dans le domaine de la propriété intellectuelle. De plus, avec le développement technologique et l'avènement d'internet comme vecteur centrale de l'activité humaine moderne, de nouveaux langages informatiques apparaissent chaque jour avec leurs spécificités, leurs objectifs, leurs syntaxes ...

L'enseignement NF11 aborde les notions théoriques nécessaires à la compréhension, à l'analyse, à la conception et à la compilation des langages de programmation et d'échange de l'information. À l'aide d'outils mathématiques et d'algorithmes existants et éprouvés, elle traite les phases d'analyses lexicale, syntaxique et sémantique. Son objectif est également de présenter les différents types de grammaires, régulières et hors contexte, ainsi que les automates associés aux analyseurs.

Pour remplir ses différents objectifs pédagogiques, l'UV propose aux étudiants la réalisation d'un projet qu'ils doivent mener au cours de leur semestre universitaire. Ainsi, en se basant sur le langage Logo, ils doivent réaliser un interpréteur lexical et syntaxique qui sera capable de prendre en entrée un ensemble d'instructions et qui devra les exécuter en gérant toutes les subtilités et spécificités du langage.

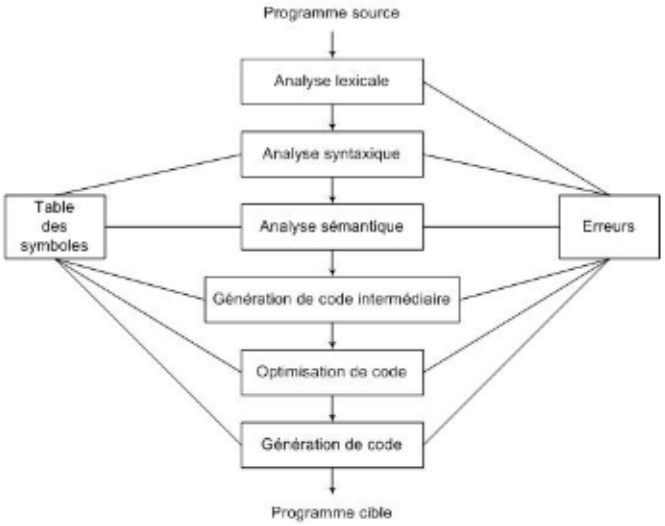

FIG. 1 – Les différentes étapes du code source au programme exécutable

1 Le projet

1.1 Aperçu

L'objectif de cette partie n'est pas de réécrire le cours de l'UV qui est disponible dans le polycopié associé ou les diapositives de présentations, mais seulement de citer et de situer simplement certains éléments importants qui ont été abordés au cours du projet.

1.1.1 Composantes du projet

Ainsi, ce dernier est constitué d'un ensemble de fichiers exploitant les facilités du *langage Java* et *ANTLR*, acronyme de ANother Tool for Language Recognition, un framework libre de construction de compilateurs, qui permet de générer du code exécutable à partir de fichiers de « grammaire » que nous détaillerons par la suite.

Le projet de base est donc constitué de trois parties principales :

- Une interface graphique, qui nous permet d'observer et de valider le travail de développement des grammaires que nous réalisons ;
- Un traceur, qui réalise les opérations demandées dans les instructions du langage LOGO ;
- Un ensemble de fichiers gérant les différentes phases d'analyses nécessaires ;

1.1.2 Les phases d'analyse

La partie principale sur laquelle nous nous sommes concentrés est donc la partie d'analyse du langage. L'objectif était ainsi de réaliser une grammaire permettant l'analyse du langage Logo de manière descendante.

Globalement, les différentes phases d'analyse sont :

- l'analyse lexicale, qui vise à reconnaitre des ensembles de caractères du flux d'entré afin d'identifier des mots clés et des catégories de « mots », appelés symboles terminaux de la grammaire ;
- l'analyse syntaxique, qui quant à elle a pour objectif d'assurer un enchainement logique de ces catégories de mots selon une grammaire prédéfinie ;
- et l'analyse sémantique, qui s'effectue en parallèle de l'analyse syntaxique et qui vérifie le « sens », la cohérence des enchainements des instructions reconnues par l'analyse syntaxique.

1.1.3 Processus d'exécution

Concrètement, à l'exécution d'un programme ou d'une suite d'instruction en Logo, notre système va réaliser deux « passages ».

Lors du premier passage seront effectuées les différentes phases d'analyses qui construiront ainsi un Arbre Syntaxique Abstrait. Si aucune erreur au cours des analyses n'a été identifiée, un second passage sera effectué. Ce dernier consistera à parcourir cet AST afin d'associer des méthodes implémentées et du code exécutable aux instructions Logo reconnues.

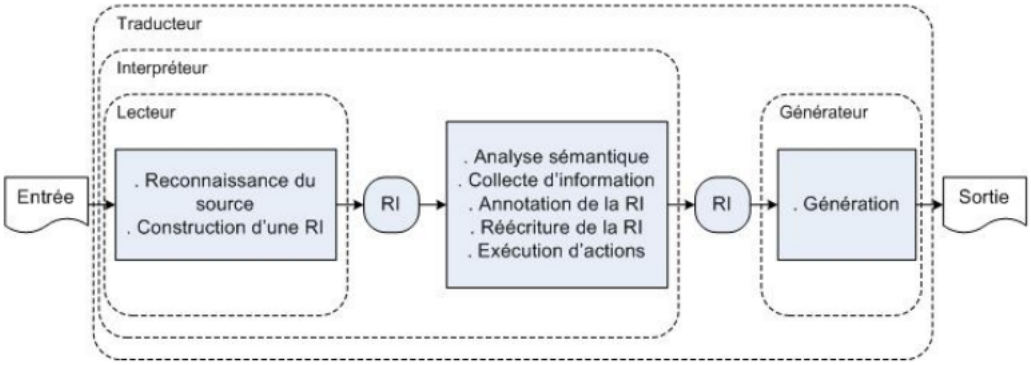

FIG. 2 – Processus générale de fonctionnement

Processus générale :

- 1re passe (lecteur)
- 2e passe (interpréteur)

1.2 Le langage Logo

Le Logo est un langage de programmation qui se veut simple et accessible, mais également, riche, structuré et fonctionnel.

La simplicité et l'accessibilité sont deux principes majeurs qui ont motivé sa conception, l'objectif étant qu'il soit très facile d'utilisation et qu'il puisse exploiter la puissance de l'outil informatique dans les tâches d'enseignement. De ce fait, la simplicité de mise en œuvre de ce langage est étonnante : quelques minutes suffisent pour en comprendre la logique et se lancer dans la création de projets personnels.

Pourtant, malgré cette simplicité apparente, Logo est un véritable langage de programmation qui permet d'aborder des concepts fondamentaux de l'informatique tels que les procédures et la récursivité.

De plus, nous pouvons distinguer les types d'instruction suivants :

- Instruction simples et commandes de base
- Les structures de contrôles
  - La répétition
  - L'alternative
  - La boucle « tant-que »
- Les fonctions de calcule
  - La primitive « Ecris »
  - Les expressions mathématiques infixe
  - Les expressions logiques
- Les variables
- Les procédures et fonctions
  - Sans paramètres
  - Passages de paramètres
  - Fonction (retour de valeur)

Chaque type d'instruction fera l'objet d'un exemple de fonctionnement et de traitement dans la suite du rapport.

1.3 La grammaire et les éléments de traitement

Au niveau de la gestion des grammaires, nous disposons de deux fichiers : *Logo.g* et *LogoTree.g*, qui symbolisent clairement les actions effectuées dans chacune des « passes » citées précédemment.

1.3.1 Lexer

Comme cité au par avant, l'analyse lexicale a pour objectif de reconnaitre des chaines de caractères appelés des TOKENs ou symboles terminaux de la grammaire. Ces TOKENs sont ainsi définis de la manière suivante :

Déclaration simple de TOKENs - Logo.g

```
1 // Commandes simples
```

```
  AV = 'AV' ;
  TD = 'TD' ;
  TG = 'TG' ;
```

Tous ces TOKENs doivent être uniques et peuvent appartenir à une même unité lexicale :

TOKEN rassemblés en Unité lexicale - Logo.g

```
ID : ('A'..'Z' | 'a'..'z')( ('A'..'Z' | 'a'..'z') | ('0'..'9'))* ;
WS  :   (' '|'\t'|('\r'? '\n'))+ { skip(); } ;
```

Nous pouvons même remarquer que ANTLR nous permet même d'utiliser des notations usuellement utilisées dans les expressions régulières telles que les opérateurs de quantification : '*', '+' et '?'.

De même, nous pouvons anticiper quelque peu la suite du rapport en remarquant une action associée au token `WS : { skip() ; }`, nous y reviendrons.

1.3.2 Parser

Le « Parser » réalise l'analyse syntaxique et repose sur une grammaire de type hors contexte. À partir de règles définies, il construit l'AST qui sera exploité dans la seconde « passe ». Encore une fois, nous pouvons citer rapidement quelques manières d'insérer des TOKENs dans l'arbre syntaxique abstrait :

Mise en arbre AST - Logo.g

```
 // 1 - Utilisation du token fictif PROGRAMME
liste_instructions -> ^(PROGRAMME liste_instructions)
;

// 2 - insertion simple en feuille de \og{}instruction\fg{} qui est en fait une regle ->
    nouvelle branche
liste_instructions :
  (instruction)+
;

// 3 - Creation d'une branche avec le TOKEN REPETE en tant que racine (grace au caractere
    '^')
repete  :
 REPETE^ expr loop_liste
;
```

L'utilisation de TOKEN fictif nous permet de résoudre certains problèmes de récursivité à gauche dont est sensible l'analyse descendante de type LL utilisée par ANTLR, mais cela nous permet également de générer des « branches » d'instructions qui seront plus facilement accessibles lors du parcours d'arbre et de l'interprétation de celles-ci comme pour le cas des procédures ou des instructions de type structurées :

Utilisation d'un TOKEN imaginaire- Logo.g

```
instructions :
  liste_instructions -> ^(INST liste_instructions)
;

procedure :
POUR^ ID param_list? Instructions FIN'!'
;
```

La syntaxe ANTLR nous permet également de réaliser une factorisation simple à gauche des instructions :

Factorisation à gauche simple- Logo.g

```
instruction :
  ( AV^
  | TD^
  | TG^
  | REC^
  | FCAP^
  | FCC^
  | ECRIS^)
  expr
```

Et de ne pas positionner dans l'arbre des TOKENs qui nous servent uniquement à lever l'ambigüité et d'indéterminisme de certaines règles (à l'aide du caractère '!').

Ignorance de TOKENS - Logo.g

```
// Loops
loop_liste
  :
  '['^ liste_instructions ']'!
;
```

Cela fût notamment le cas pour l'appel de procédure où nous avons eu besoin de rajouter un token ';' dans la déclaration pour éviter des problèmes de récursivité de type LL* sans activer l'option de « backtracking ».

Nécessité d'un caractère de différentiation supplémentaire - Logo.g

```
ID^ (p=paramValue_list)? ';'!
```

1.3.3 Treewalker

Les règles et leur syntaxe pour le parcours de l'Arbre Syntaxique Abstrait sont très proches de celle du « parseur ». Les principales différences avec ces dernières sont la récupération des informations

stockées dans l'arbre, les actions associées, et l'utilisation d'attributs hérités ou synthétisés.

Exemple d'actions associées - LogoTree.g

```
// Action realisee a l'issue du parcours de la branche issue du noeud racine RET
retour :
^(RET a=expr) {contexte.setFunctionValue($a.v);}


// retour de l'attribut 'value' pour la variable 'cap'
cap returns [double value] :
  CAP {
    $value = traceur.cap();
    //Log.appendnl("cap : "+value);
  }
;
```

Une particularité intéressante que nous avons pu utiliser pour les procédures et les instructions structurées est la possibilité d'ignorer une branche durant le parcours en profondeur d'abord et de mémoriser son indice dans l'arbre pour y revenir ultérieurement.

Ignorance de branches - LogoTree.g

```
// On ignore les branches qui peuvent suivre le TOKEN 'INST'
instructions :
  ^(INST (.)+ )
;


// Ignore les instructions mais on memorise leurs positions via l'action input.mark()
tantque
: ^(TANTQUE ({mark_x = input.mark();} .) ({mark_list = input.mark();} .) );
```

1.3.4 Table des symboles

Le langage Logo permet la manipulation de variables. Les variables sont des éléments auxquels il est possible d'attacher des valeurs et de les consulter à tout instant dans les instructions suivantes du programme.

Puisque le langage sur lequel nous nous basons pour réaliser notre analyseur est le Java, nous utilisons une « HashMap » pour implémenter le concept de table des symboles.

Une « HashMap » est en fait un tableau associatif qui contient des couples du type (clé => valeur). Ainsi, en connaissant le nom de la variable, il est très facile de retrouver la valeur associée.

La « HashMap » possède également différentes propriétés qui sont très proches des celles des variables du langage. La principale étant la notion de portée et de contexte. Dans ce principe, une variable avec un certain nom ne peut être déclarée deux fois et avoir deux valeurs différentes à un même instant. L'ajout d'une nouvelle valeur pour une variable existante écrasera la valeur précédente. Nous aborderons la gestion de la portée dans la section suivante.

D'un point de vue purement technique, la « HashMap » représentant un contexte, ou une portée, peut contenir des couples du type <String, Object>. De ce fait, elle associe à un nom n'importe quel objet eu langage Java. Les principaux objets utilisés sont principalement les « Doubles » pour les variables et des `LogoProc`, que nous allons voir, pour les procédures et fonctions.

1.3.5 Les structures utiles

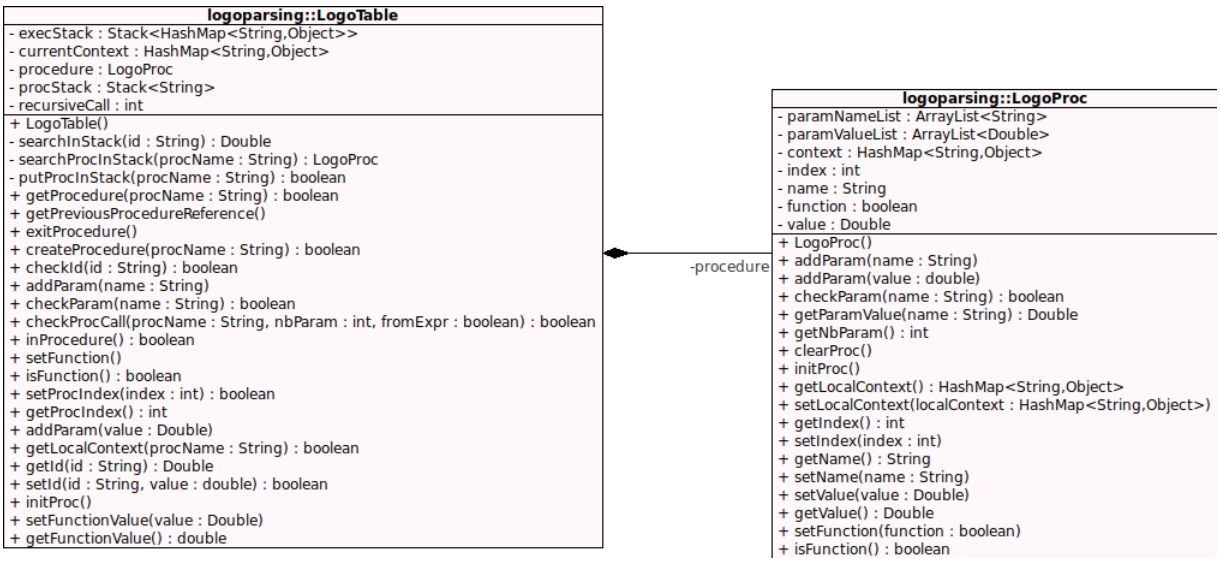

FIG. 3 – Représentation UML des classes utiles, rajoutées au projet

En essayant d'être le plus concis et le plus clair possible, nous allons expliquer les structures que nous avons utilisées au cours du projet. Nous pouvons ainsi remarquer que nous utilisons deux structures particulières pour la gestion des portées de variables et d'instructions spécifiques.

`LogoTable` est en fait une structure constituée de deux piles, de deux références et d'une variable dédiée à compter le nombre d'appels des procédures ou fonctions afin de gérer la récursivité du langage. Parmi les deux piles citées, une première nommée `execStack` contient les différents « contextes » utilisés au cours du programme, les contextes étant de type « HashMap ». La seconde pile `procStack` permet de conserver les appels imbriqués ou récursifs des procédures ou fonctions. Ainsi, ces deux piles nous permettent de maintenir un certain ordre de séquence. `LogoTable` n'est instanciée qu'une seule fois et possèdes des attributs « référence » vers le sommet de la pile d'exécution, le contexte courant, et vers la procédure en cours. Lorsque des procédures sont déclarées, des objets `LogoProc` sont crées et stockés dans le contexte global du programme, autrement dit, le contexte en fond de pile `execStack`.

Une particularité de notre implémentation réside dans le fait que nous ne considérons pas l'instruction « LOCALE » et avons choisi que toute déclaration de variable soit réalisée dans le contexte « courant », et toute déclaration de fonctions et de procédures dans le contexte global. De ce fait, les variables sont toujours locales aux fonctions et procédures si elles sont déclarées à ce niveau.

`LogoProc`, est une classe qui représente une procédure ou une fonction. Elle possède comme éléments deux « ArrayList », qui nous permettent de gérer les paramètres éventuels (paramNameList et paramValueList), une « HashMap » représentant son contexte local, et dont la référence sera mise en pile lors de l'appel, ainsi qu'autres attributs comme l'index des instructions à réaliser (voir TreeWalker et le `input.mark()`), un nom, une valeur de retour si c'est une fonction et une variable booléenne indiquant s'il s'agit d'une fonction ou pas.

Pour chaque structure, nous avons essayé de définir des fonctions génériques et utiles, qui nous permettent de réaliser les actions nécessaires au bon fonctionnement de l'analyseur.

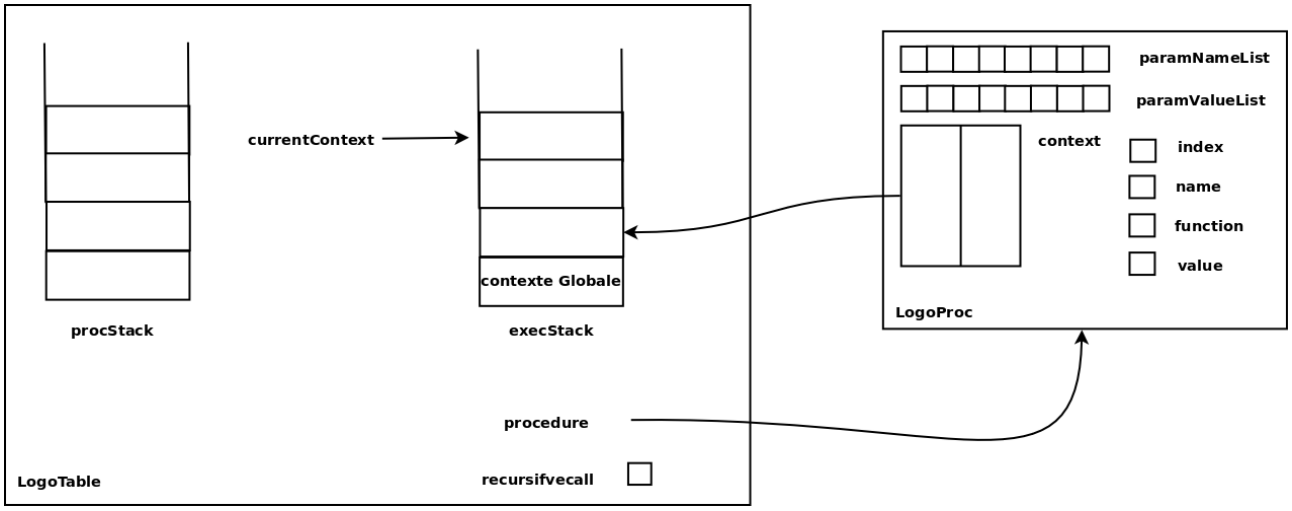

FIG. 4 – Schéma logique des structures utiles

1.3.6 Fonctionnement et interactions

Maintenant que l'organisation générale du projet et ses éléments sont définis, nous allons voir de quelle manière nous les relions.

La fonction `Main` de notre application réalise l'appel à l'environnement et l'interface graphique nous permettant de contrôler l'implémentation de l'analyseur du Logo. Dans cette interface, la méthode `runParser()` va instancier les différents objets représentant les analyses.

`runParser()` instancie ensuite un objet de type `LogoTable`, dont la référence sera passée en paramètre à chacun des analyseurs dans l'ordre d'exécution.

Au cours de la première « passe », les différentes analyses peuvent être amenées à générer des erreurs qui viendront impacter la suite de l'exécution. Un paramètre booléen nommé `valide` sera récupéré et

conditionnera le passage à la seconde « passe ».

Une partie du code de la fonction runParser() - LogoFrame.java

```
/*-- LEXER + PARSER : Analyses en premiere passe --*/
// 1 - Instantiation Lexer
ANTLRStringStream str = new ANTLRStringStream(program);
LogoLexer lexer = new LogoLexer(str);

// 2 - Instantiation Parser
CommonTokenStream tokens = new CommonTokenStream(lexer);
LogoParser parser = new LogoParser(tokens);

// Table des variables : context globale
LogoTable context = new LogoTable();

// 3 - execution Passe 1 : Lexer + Parser
LogoParser.programme_return r = parser.programme(context);


// Gestion des erreurs : Condition pour la 2emme passe
boolean cont = parser.getValide();



/*-- Tree walker : interpretation --*/
// Recuperation de l'AST
CommonTree parseTree = (CommonTree) r.getTree();
   getJASTTree().setModel(new ASTtoTreeModelAdapter(parseTree));
   expandAll();

   if (isViewVisible() && cont) {
     // 1 - Creation du noeud racine et du TreeWalker
     CommonTreeNodeStream nodes = new CommonTreeNodeStream(parseTree);
     LogoTree treewalker = new LogoTree(nodes);
     treewalker.initialize(getJLogoPane().getGraphics());

     // 2 - Passe le contexte
     treewalker.prog(context);
  }
```

Au niveau des fichiers de grammaires, nous récupérons l'objet `context` (`LogoTable`) à travers le contexte dynamique `scope` après l'avoir déclaré dans les membres :

Récupération d'objet de Java - Logo.g

```
@members{
  boolean valide = true;
  public boolean getValide(){
   return valide;
  }
  public LogoTable contexte;
}
```

```
8
9  programme[LogoTable scope]
10 @init{
11  contexte = scope;
12 }
```

En suite, au cours des analyses, nous accédons aux éléments de « contexte » de manière classique comme en Java :

Manipulation d'objet Java dans les grammaires- Logo.g

```
1  procedure
2  @init{
3  boolean test = false;
4  }
5  :
6   POUR^ ID {test=contexte.createProcedure($ID.text);} param_list? instructions {if( test ){
       contexte.exitProcedure();}}
7  ...
```

1.4 Les points spécifiques en détails

1.4.1 Gestions des variables

En logo, les variables se déclarent et s'initialisent avec l'instruction :

```
DONNE " ID valeur
```

Déclaration de variables : Au cours de la première « passe », nous cherchons uniquement à valider la syntaxe et la sémantique du code logo. Ainsi, à la déclaration d'une variable, notre fonction va d'abord vérifier la présence d'une variable ou d'une procédure / fonction avec le même identifiant dans le contexte courant. Si elle ne trouve aucune valeur, alors nous allons insérer la nouvelle variable dans la table de hachage, initialisée avec une valeur nulle (0) par défaut. Dans le cas où la méthode aurait trouvé une entrée dans la table, elle renverrait une valeur d'échec, la variable de validation de l'analyse prendrait la valeur fausse et un message d'erreur s'affichera dans la fenêtre prévue à cet effet dans l'interface graphique afin d'avertir l'utilisateur.

Déclaration de variables - Logo.g

```
1  set :
2  DONNE^ '"'!ID expr {if( !(contexte.setId($ID.text, 0)) ){
3       Log.appendnl("Conflit : fonction ou variable \"" + $ID.text + "\" deja declaree");
4       valide=false;
5     }
6     }
7  ;
```

Déclaration de variables - LogoTree.g

```
set :
  ^(DONNE ID b=expr){contexte.setId($ID.text, $b.v);}
;
```

Remarque :

Dans notre cas, nous avons seulement considéré le cas où les variables peuvent contenir des valeurs numériques provenant de l'évaluation d'expressions. Ainsi, il n'est pas possible d'utiliser et d'affecter des chaines de caractères ou des valeurs booléennes.

L'Appelle de variables : s'effectue à l'aide de l'instruction

```
:nomVariable
```

À cet instant pour gérer la notion de portée, la recherche s'effectue en trois temps :

1. Dans le premier temps, nous allons chercher dans le contexte courant ;
2. Si aucune valeur n'est trouvée, nous allons chercher dans les paramètres de la procédure/ fonction si nous sommes en présence de cette structure (vérifie la valeur de la référence de la procédure courante de `LogoTable`) ;
3. Si aucune valeur n'est toujours trouvée, alors nous remontons la pile des contextes `execStack` à la recherche d'une variable « globale ».

Nous avons ainsi implémenté deux types de méthodes :

- `checkXX`, qui vérifient la présence dans le contexte ;
- et `getXX`, qui renvoient les valeurs associées.

Appelle de variables - Logo.g

```
  // Vars - identificator
get :
  GET^ ID { if( !(contexte.checkId($ID.text)) ){
          Log.appendnl("Variable \"" + $ID.text + "\" non declaree");
          valide=false;
        }
        }
;
```

Appelle de variables - LogoTree.g

```
    | ^(GET ID) {$v = contexte.getId($ID.text);}
```

1.4.2 L'alternative

Dans le cas de l'alternative, nous utilisons les possibilités évoquées d'ignorer une branche de l'arbre au cours du parcours initial et d'y revenir par la suite en sauvegardant son index.

Parcours à postériori, utilisation d'index - LogoTree.g

```
// donc on push l'indice sauvegarde lors du parcours d'arbre
((CommonTreeNodeStream)input).push(mark_list);

// Appelle de la regle d'interpretation de l'arbre
liste_instructions();

// on pop pour retrouver l'indice normal
(CommonTreeNodeStream)input).pop();
```

La particularité de l'instruction d'alternative est que le second bloc d'instructions est facultatif. Pour ce faire, nous initialisons le marqueur d'index à -1. Si la récupération de l'index a eu lieu grâce à la méthode `input.mark()` alors nous pouvons le traiter. Dans le cas contraire, l'interpréteur ne l'a pas trouvé.

Bloc optionnel - LogoTree.g

```
alternative
@init{
int mark_list = 0;
int mark_list2 = -1;
}
: ^(SI x=boolExp ({mark_list = input.mark();} .) ({mark_list2 = input.mark();} .)?)
{
  ...

  if (mark_list2 != -1){
    // Traitement
  }
}
```

1.4.3 Les boucles

En ce qui concerne la gestion des boucles, la partie la plus importante se situe dans la partie Java de la grammaire d'interprétation. En effet, leur syntaxe est très clairement identifiable avec des TOKENs uniques représentant le type d'instruction et une délimitation des instructions concernées grâce aux TOKENs '[' et ']'. Nous utilisons également la création d'une branche contenant toutes les instructions associées grâce à l'utilisation d'un TOKEN imaginaire en racine.

Les boucles - Logo.g

```
// Loops
loop_liste
```

```
  :
  '['^ liste_instructions ']'!
;

repete  :
 REPETE^ expr loop_liste
;

alternative :
 SI^ boolExp loop_liste loop_liste?
;

tantque  :
 TANTQUE^ boolExp loop_liste
;
```

Le traitement est très proche de celui de l'alternative où nous mémorisons les index des nœ uds racines des instructions et des éléments auxquels nous devons accéder, comme les conditions de boucle.

Les boucles - LogoTree.g

```
tantque
@init{
int mark_x = 0;
int mark_list = 0;
boolean x=false;
}
: ^(TANTQUE ({mark_x = input.mark();} .) ({mark_list = input.mark();} .) )
  {

    // ACCES A L'INVARIANT DE BOUCLE
    ((CommonTreeNodeStream)input).push(mark_x);
    x=boolExp();
        ((CommonTreeNodeStream)input).pop();


    while(x == true ){

    ... Traitement

      // On recupere la nouvelle valeur de X
      ((CommonTreeNodeStream)input).push(mark_x);
      x=boolExp();
      ((CommonTreeNodeStream)input).pop();
    }
  }
  ;
```

1.4.4 Le passage d'attributs dans les procédures

La problématique dans le passage d'attributs aux procédures et fonctions est la suivante :

Lors de la déclaration de la procédure, l'instruction Logo fournit une liste d'identifiant de variable. Cependant, celles-ci ne seront initialisées que lors des différents appels de la procédure, où nous n'aurons cette fois-ci qu'une liste de valeurs. De plus, avec la structure de « Hashmap » mise en place, nous ne pouvons accéder à une variable que si nous connaissons son identifiant.

Nous avons ainsi deux objectifs :

1. La vérification de l'arité de la procédure lors de l'appel pendant la phase d'analyse ;
2. La gestion de la récursivité lors de l'interprétation et la génération du code exécutable.

Pour résoudre ces problèmes et atteindre nos objectifs, un objet `LogoProc` et deux tableaux de structures linéaires : « ArrayList ».

Lors de la phase d'analyse et de la **déclaration de la procédure**, nous appelons une méthode de création de procédure, ensuite nous passons à la déclaration des paramètres et l'évaluation des instructions.

Déclaration de procédure - Logo.g

```
  procedure
@init{
boolean test = false;
}
:

 POUR^ ID {test=contexte.createProcedure($ID.text);}   // 1 -
   param_list?              // 2 -
   instructions {if( test ){contexte.exitProcedure();}}  // 3 -
 FIN

 {
   // 1 - Traitement des erreurs
   if( !(test) ){
      Log.appendnl("Function \""+$ID.text+"\" or VAR already declared in this scope");
      valide=test;
   }

 }
;


param_list
  :
  (':'ID {contexte.addParam($ID.text);} )+ -> ^(PARAM ID+)
  ;
```

`createProcedure()` :
Cette méthode vérifie que la procédure n'existe pas encore. Si c'est le cas, elle crée un nouvel objet `LogoProc` dans le contexte global du programme Logo en associant l'objet au nom de l'identifiant de la procédure. Sinon, elle avertit l'utilisateur du conflit de nom.

Pour permettre l'évaluation des instructions, elle charge également le contexte de la procédure en pile et déplace la référence du contexte courant en sommet de pile. Elle place également la référence de la procédure courante sur l'objet dans le contexte global de manière à accéder à ces éléments.

Déclaration des paramètres :
La déclaration des paramètres consiste à l'insertion des identifiants dans le tableau correspondant.

`exitProcedure()` :
La sortie de la procédure consiste à dépiler le contexte local de la procédure, replacer la référence du contexte courant en sommet de pile et réinitialiser la référence sur la procédure.

À l'appel de la procédure, nous insérons les paramètres en feuille du sous-arbre représenté par l'identifiant de procédure et nous utilisons un attribut à la règle pour calculer leur nombre. Ce dernier est transmis à une méthode `checkProcCall` qui se chargera de vérifier si le nombre de valeurs est bien égal à la dimension du tableau d'identifiants des variables en paramètres.

À noter que nous utilisons également un paramètre pour la règle d'appel de procédures. Ce paramètre a pour objectif de différencier les procédures des fonctions. Par défaut, ce paramètre possède la valeur « fausse ». Mais si l'appel est effectué à partir de la règle qui représente les expressions mathématiques, alors il s'agit d'une fonction. `checkProcCall` ira modifier le paramètre de l'objet `LogoProc` pour qu'il soit considéré comme une fonction.

Appel de procédure - Logo.g

```
1  paramValue_list returns [int nbParam]
2  @init{
3  $nbParam = 0;
4  }
5    :
6    (expr {++$nbParam;})+
7    ;
8
9  procCall [boolean fromExpr] :
10 ID^ (p=paramValue_list)? ';'!
11
12   { if( !(contexte.checkProcCall($ID.text, $p.nbParam, $fromExpr)) )
13   { valide=false; }
14   }
15 ;
```

Phase d'interprétation : Lors de la déclaration, nous ne faisons que sauvegarder l'indice du sous arbre représentant les instructions à réaliser dans l'index de la procédure.

Pour l'appel,

1. Nous chargeons le contexte local de la procédure et recherchons sa référence dans le contexte global ;
2. Nous enregistrons ensuite les valeurs des paramètres de la même manière que les identifiants réalisé en première « passe ». Pour garantir ensuite le fonctionnement instructions, nous associons les variables à leur valeur en les insérant dans le contexte local de la procédure ;
3. Nous chargeons l'index du sous arbre d'instructions à l'image de ce qui est fait dans les boucles ou l'alternative ;
4. En fin d'exécution, si la procédure est une fonction, on valut son attribut correspondant ;
5. Nous finissons en replaçant les références et les contextes dans les états qu'ils possédaient avant l'appel ;

Appel de procédure - LogoTree.g

```
paramValue_list
: (a=expr {contexte.addParam($a.v);})+
;


procCall returns [Double value]
@init{
int mark_instructions = -1;
int mark_param_list = -1;
int i=0;
}:
 ^(ID ({mark_param_list = input.mark();} (.)*) )
 {
   // 1 - Recupere le contexte local declare en 1ere passe
   if( !contexte.getLocalContext($ID.text) ){
    return null;
   }

   // 2 - initialisation des paramettres
   if(mark_param_list != -1){
    ((CommonTreeNodeStream)input).push(mark_param_list);
  paramValue_list();
  ((CommonTreeNodeStream)input).pop();

  contexte.initProc();
  //System.out.println("procCall : \""+$ID.text+"\" initialisation param OK ");
   }else {System.out.println("procCall : \""+$ID.text+"\" pas de paramettres ");}


   // 3 - Recupere l'index des instructions
   mark_instructions = contexte.getProcIndex();
   //mark_instructions = contexte.getId($ID.text).intValue();
   //System.out.println("procCall : Appelle de \""+$ID.text+"\" mark_liste = "+
       mark_instructions);


```

```
36    // 4 - Execution des instuctions
37    ((CommonTreeNodeStream)input).push(mark_instructions);
38    liste_instructions();
39    ((CommonTreeNodeStream)input).pop();
40
41
42    // 5 - Si procedure = fonction => retourne une valeur
43    if(contexte.isFunction()){
44     $value = contexte.getFunctionValue();
45    }
46    else {$value = null;}
47
48
49    // 6 - Retour au contexte (n+1)
50    contexte.exitProcedure();
51  }
52 ;
```

1.4.5 Le retour de valeur des fonctions

Nous avons déjà évoqué en bonne partie la gestion des fonctions étant donné qu'en Logo, ce ne sont que des procédures possédant une instruction de retour de valeur. De ce fait, elles peuvent être appelées en tant qu'expression mathématiques.

Ainsi, au niveau de la phase d'analyse, nous pouvons simplement souligner l'importance sémantique, où nous vérifions si l'appel de l'instruction de retour s'effectue bien au sein d'une procédure. Si tel est le cas, alors nous fixons la procédure en question en tant que fonction.

instruction RET - Logo.g

```
1  retour
2 @init{
3 boolean test = false;
4 }
5 :
6   RET^ expr
7   {
8   test=contexte.inProcedure();
9   if( !test){
10    valide=test;
11    Log.appendnl("Warning ! Return expr out of function");
12  }
13  else{
14    // Si oui -> la procedure est une fonction
15    contexte.setFunction();
16  }
17
18  }
19  ;
```

Pour l'interprétation, nous n'avons qu'à fixer la valeur de l'attribut prévu à cet effet :

instruction RET - LogoTree.g

```
retour :
^(RET a=expr) {contexte.setFunctionValue($a.v);}
  ;
```

1.4.6 Gestion de la récursivité

De même, la gestion de la récursivité a déjà été évoquée dans les parties précédentes. Nous pouvons néanmoins ajouter que pour gérer complètement la récursivité, il est nécessaire de vider le tableau des valeurs des paramètres en début de chaque appel de procédure. De ce fait, nous conservons la correspondance avec le tableau des identifiants et les index.

L'utilisation d'un contexte local aux procédures et l'initialisation des paramètres en tant que variable de ce contexte permet également de ne pas écraser ces valeurs et d'obtenir un effet de bord indésiré.

De plus, pour éviter les appels récursifs infinis et un débordement des nos piles, nous avons utilisé une variable compteur qui est incrémenté à chaque appel et décrémenté à chaque sortie. La valeur limite du nombre d'appel récursif est fixée à 500.

1.4.7 Éléments non traités

- Instruction STOP ;
- Instruction LOOP ;
- Le traitement des chaines de caractères et des expressions booléennes pour les valeurs associées aux variables ;
- La gestion de ces éléments dans l'instruction ECRIS ;

2 Exemples

Quelques exemples :

2.1 Instruction simples et commandes de base

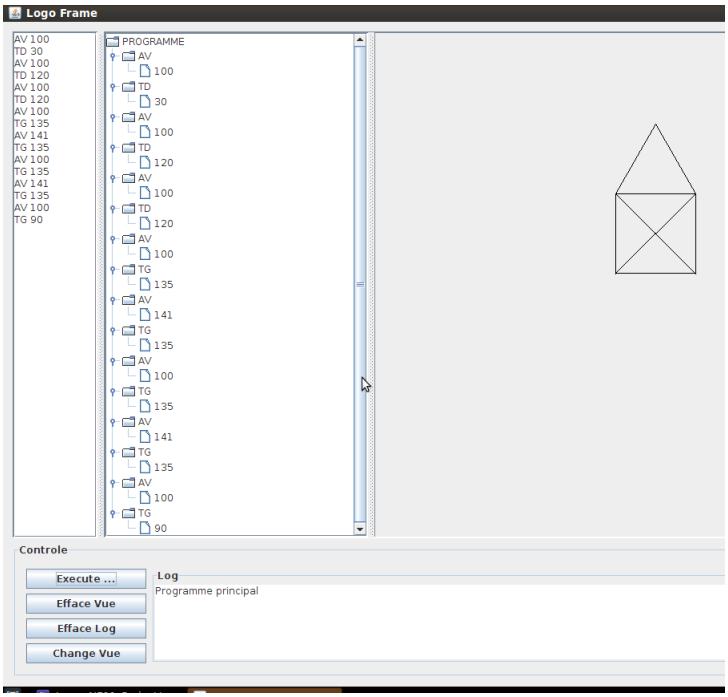

FIG. 5 – Instruction simples et commandes de base

2.2 Les structures de contrôles

2.2.1 L'alternative

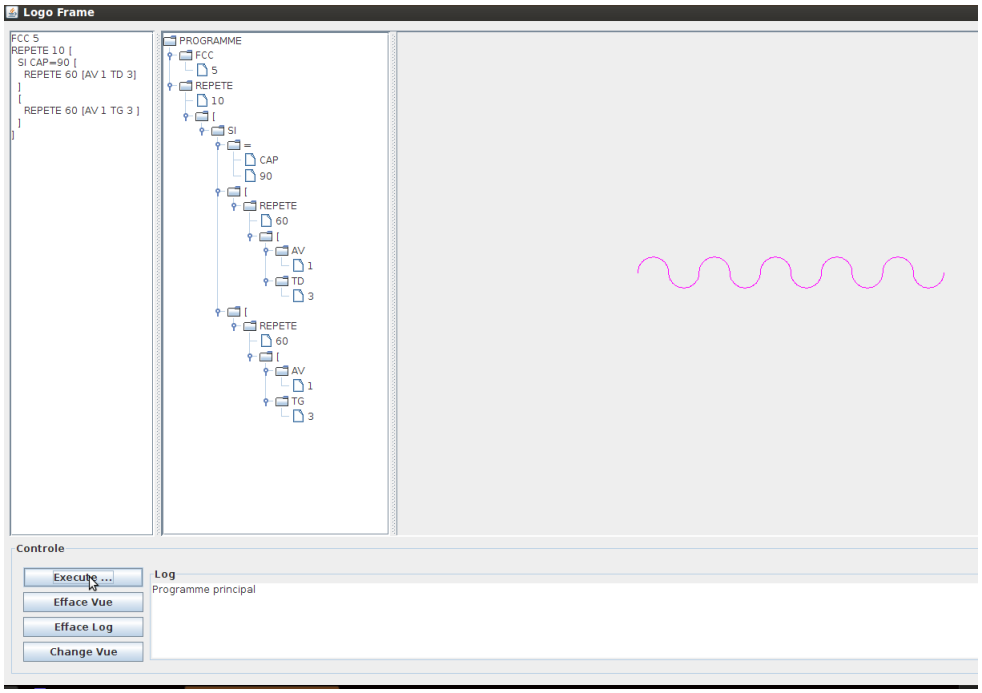

FIG. 6 – L'alternative

2.2.2 La répétition

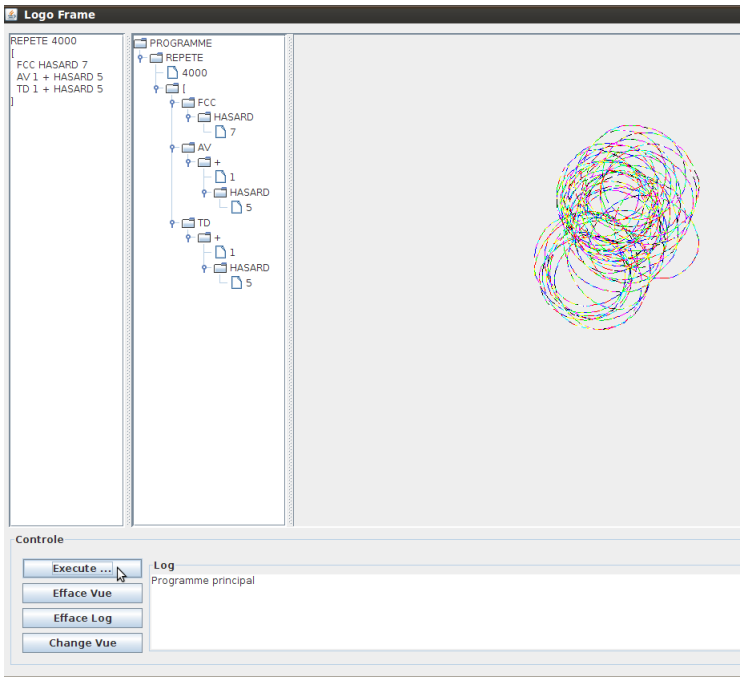

FIG. 7 – Répitition + hasard

2.2.3 La boucle TANTQUE

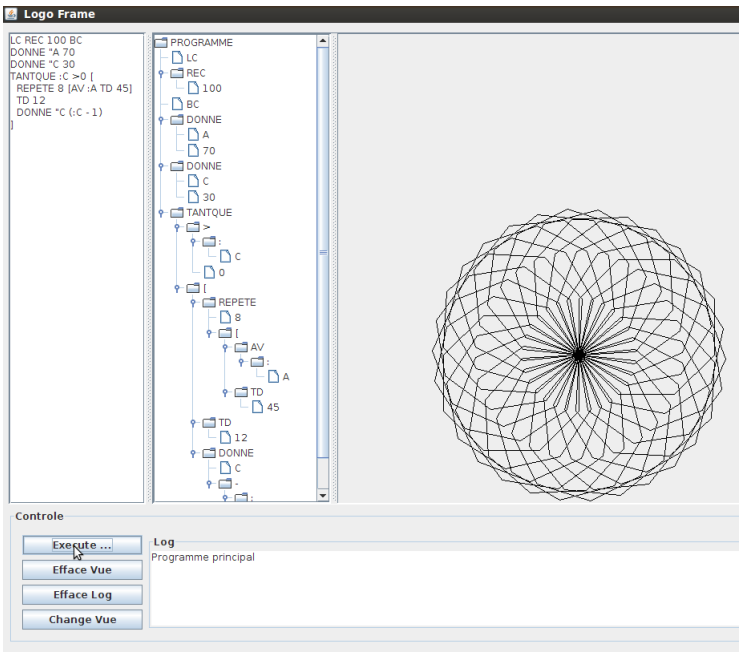

FIG. 8 – Boucles imbriquées

2.3 Calculer avec Logo

2.3.1 La primitive ECRIS

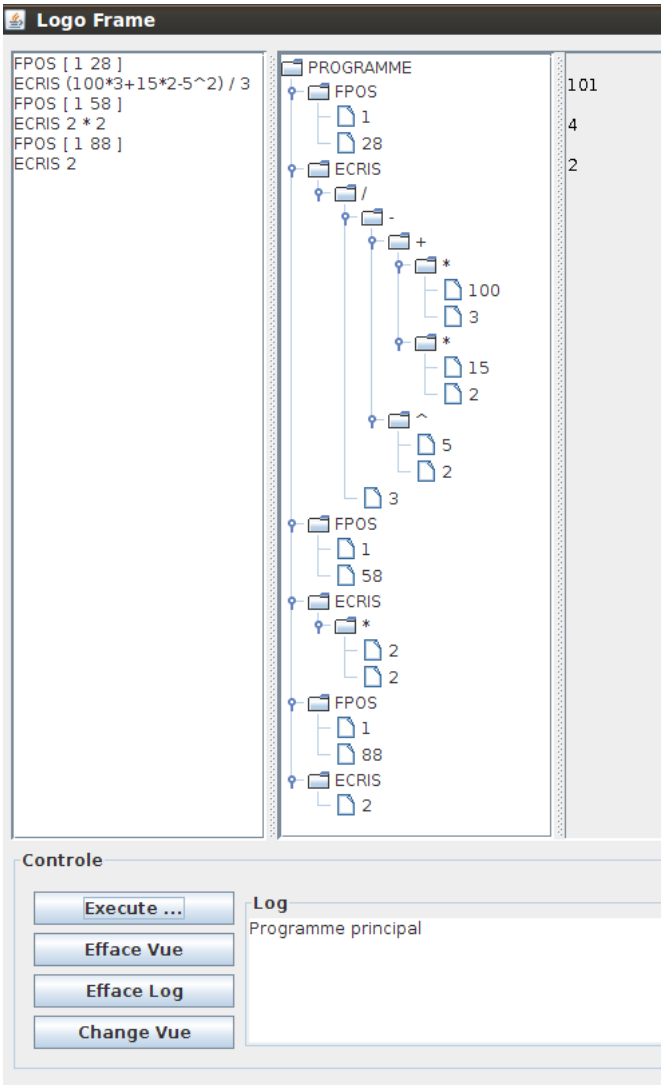

FIG. 9 – ECRIS d'une expression mathématique

2.4 Procédures et fonctions

2.4.1 Sans paramètre

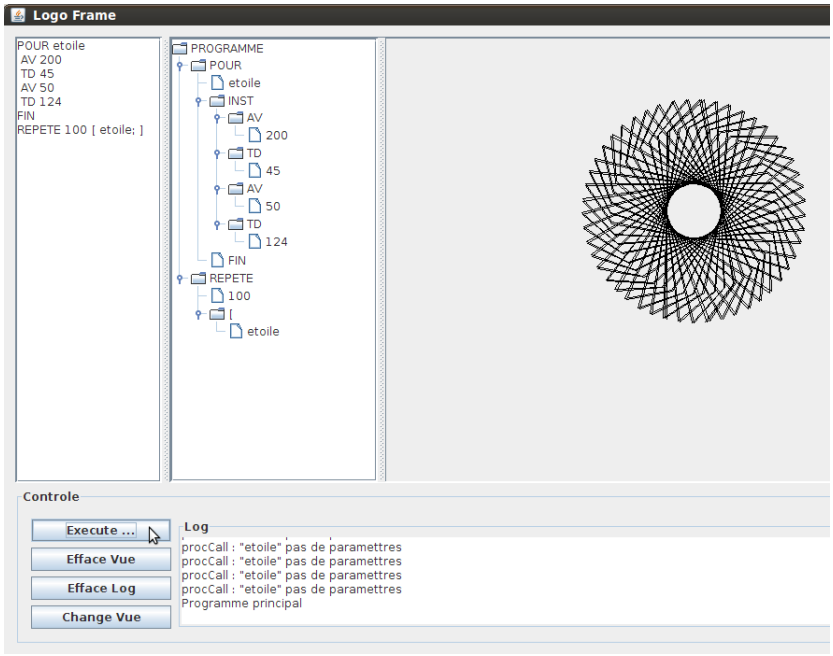

FIG. 10 – Procédure sans paramètres : Étoile

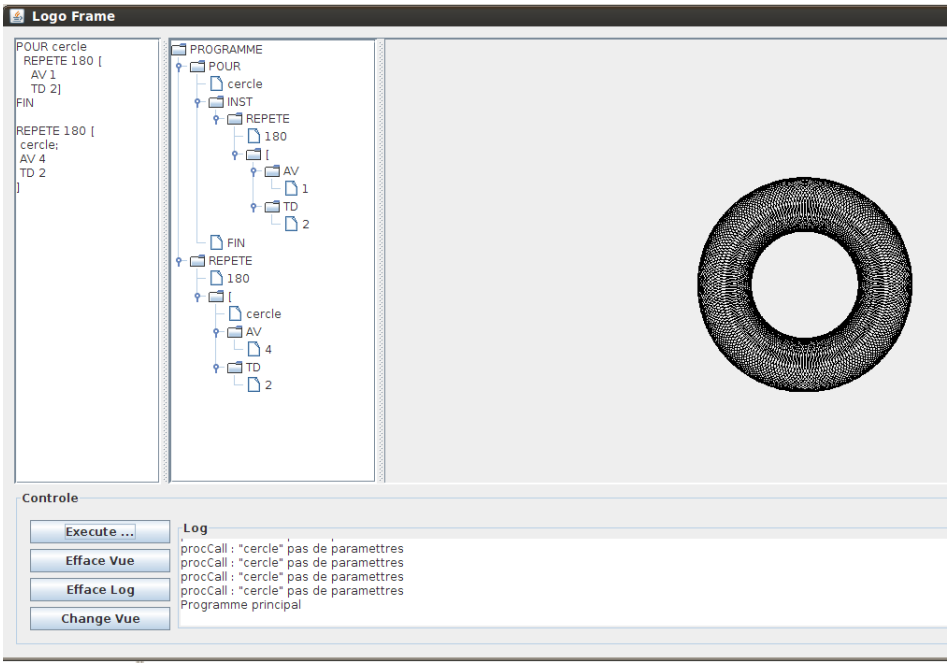

FIG. 11 – Procédure sans paramètres : Écercle

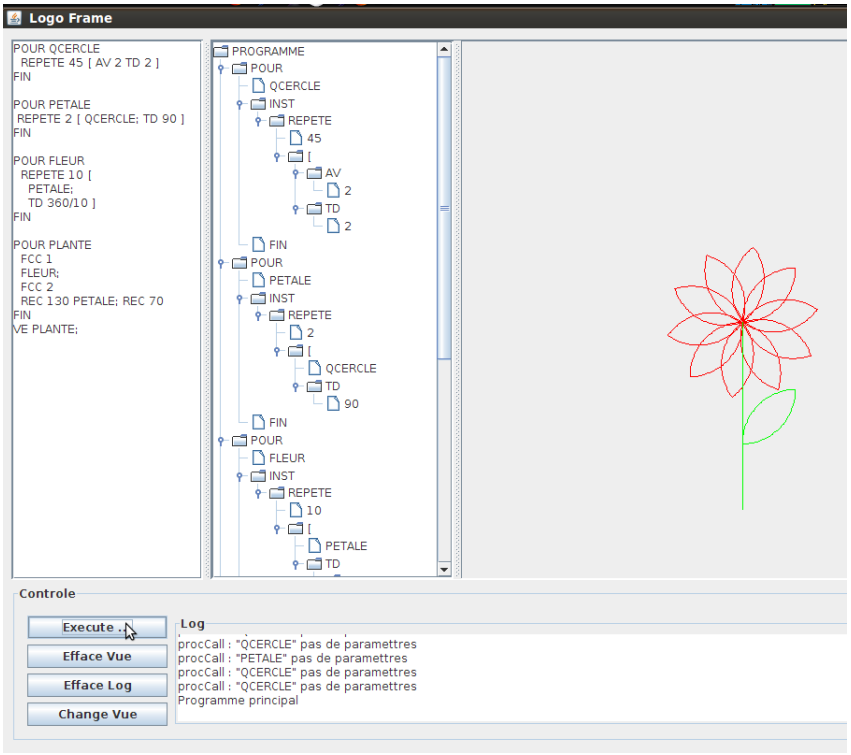

FIG. 12 – Procédure sans paramètres : Fleur

2.4.2 Passages de paramètres

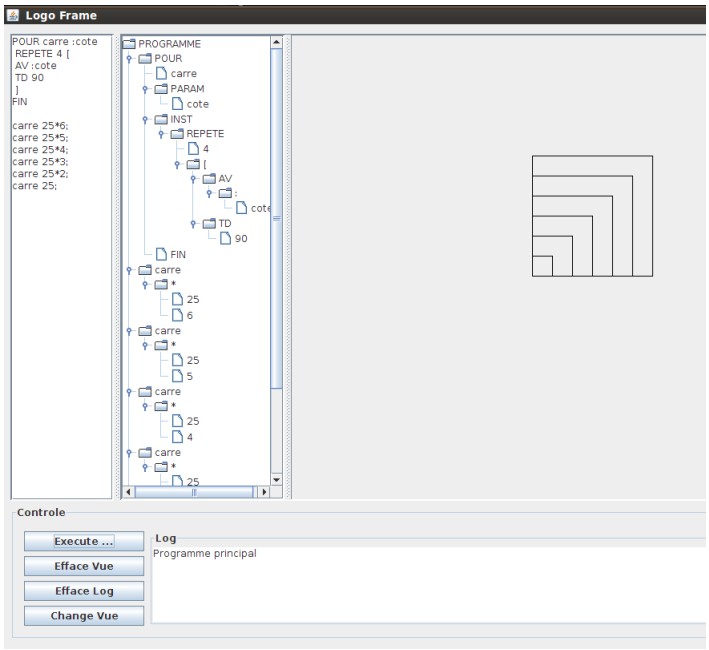

FIG. 13 – Procédure avec paramètres

2.4.3 Fonction (retour de valeur)

FIG. 14 – Fonction carré(x)

2.4.4 Les contrôles et messages d'erreur

FIG. 15 –

2.4.5 Les appels récursifs

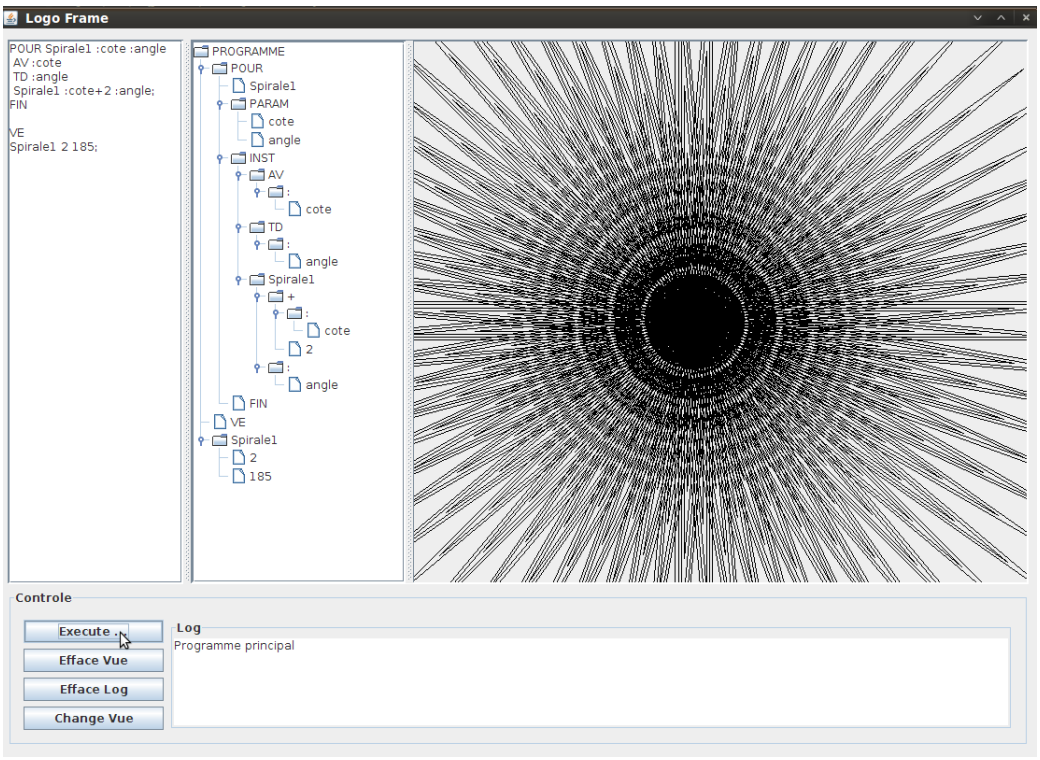

FIG. 16 – Appels récursif avec atteinte de la limite

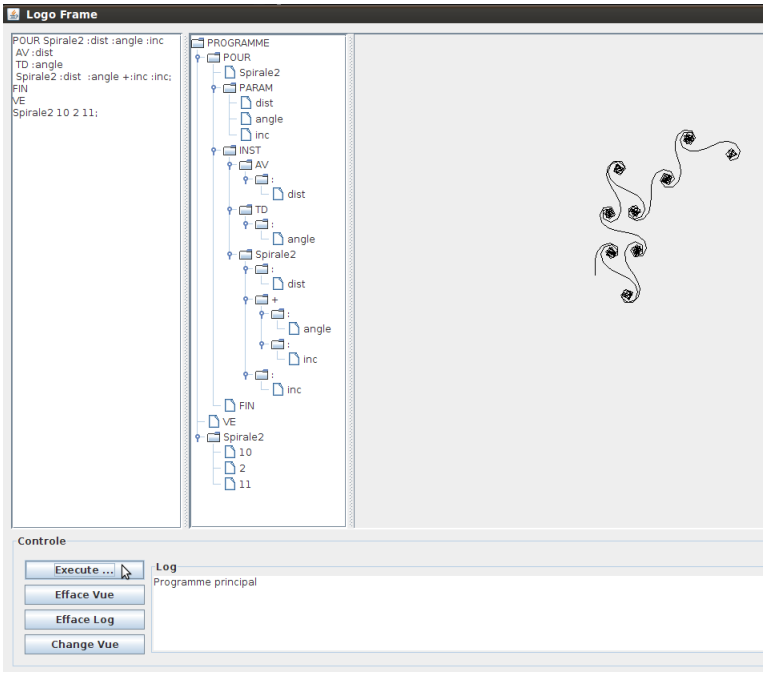

FIG. 17 – Appels récursifs - Frise

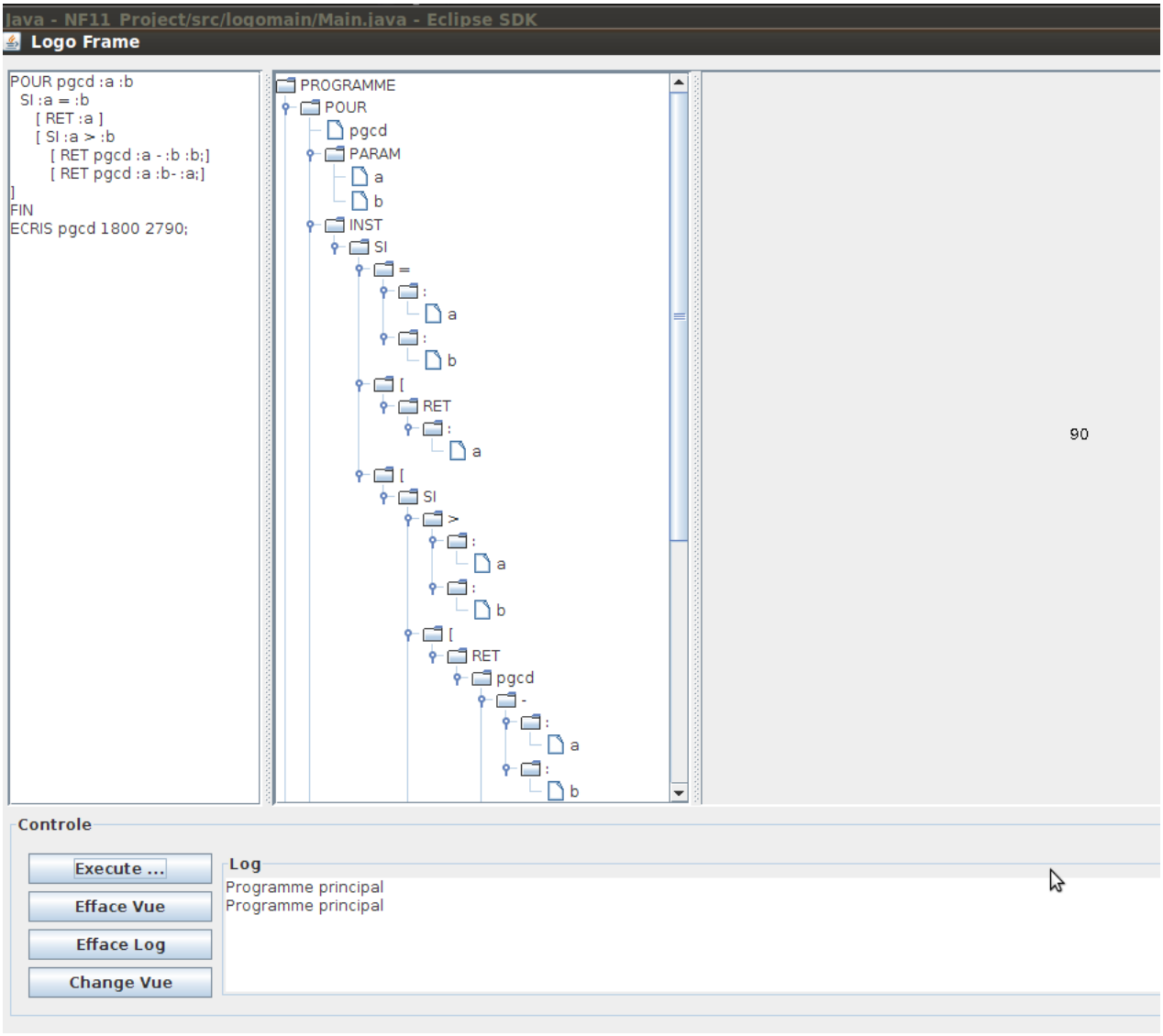

FIG. 18 – Fonction récursive - pgcd (a, b)

Conclusion

A l'issue de ce projet, nous pouvons dire que nous avons pris conscience de la complexité que représente l'analyse, la compréhension et l'exploitation d'un langage dans la conception d'un interpréteur ou d'un « parseur ». Nous pouvons également dire que, si nous avons apprécié la partie de codage et la découverte du langage Java, il est important de ne pas négliger le travail d'analyse du langage pour bien saisir ses subtilités et sa complexité. Le projet nous a également permi de remarquer que même si la conception d'une grammaire d'analyse déscendante semble plus intuitive, il reste cependant certains « pièges » à éviter pour que celle-ci soit efficace et fonctionnelle. D'une manière générale, l'uv était intéressante car elle nous a permis de connaitre réellement les méthodes utilisées pour bien comprendre un lanage. En effet, cela peut toujours être utile lorsqu'on cherche à manipuler de l'information sous forme textuelle, et avec l'évolution de notre quotidien qui tends à nous fournir toujours plus d'information à analyser, cela ne peut être qu'un avantage.

Annexes

.1 Logo.g

Lexer et Parser - Logo.g

```
grammar Logo;
options {
  output = AST;
}

// 1 - Lexer
tokens {
  PROGRAMME;
  // Commandes simples
  AV = 'AV' ;
  TD = 'TD' ;
  TG = 'TG' ;
  REC = 'REC' ;
  FPOS = 'FPOS';
  FCC = 'FCC';
  FCAP = 'FCAP';
  VE = 'VE';
  LC = 'LC';
  BC = 'BC';
  // Maths
  PLUS = '+';
  MOINS = '-';
  MULT = '*';
  DIV = '/';
  EXP = '^';
  SIN = 'sin';
  COS = 'cos';
  // Logical
  OR = '|';
  AND = '&';
  TRUE = 'TRUE';
  FALSE = 'FALSE';
  // Var manipulation
  ECRIS = 'ECRIS';
  DONNE = 'DONNE';
  GET = ':';
  // Structures - programmation
  REPETE = 'REPETE';
  TANTQUE = 'TANTQUE';
  SI =  'SI';
  // LOCAL = 'LOCALE'; // implicite pour les procedures
  POUR = 'POUR';
  FIN = 'FIN';
  INST;
  PARAM;

  HASARD = 'HASARD';
  CAP = 'CAP';
```

```
49    STOP = 'STOP';
50    RET = 'RET';
51  }
52  @lexer::header {
53    package logoparsing;
54  }
55  @header {
56    package logoparsing;
57    import logogui.Log;
58  }
59  @members{
60    boolean valide = true;
61    public boolean getValide(){
62     return valide;
63    }
64    public LogoTable contexte;
65  }
66
67  // fragment
68  // DIGIT : '0'..'9' ;
69  // INT : DIGIT+ ;
70  INT : ('0'..'9')+;
71  //fragment
72  //LET : ('A'..'Z' | 'a'..'z') ;
73  //LETTERS : ('A'..'Z' | 'a'..'z')+;
74  ID : ('A'..'Z' | 'a'..'z')( ('A'..'Z' | 'a'..'z') | ('0'..'9'))* ;
75  WS  :   (' '|'\t'|('\r'? '\n'))+ { skip(); } ;
76
77
78
79
80
81  // 2 - Rules : Parser
82  programme[LogoTable scope]
83  @init{
84   contexte = scope;
85  }
86  : liste_instructions -> ^(PROGRAMME liste_instructions)
87  ;
88
89
90  liste_instructions :
91    (instruction)+
92  ;
93
94
95  instruction :
96    ( AV^
97    | TD^
98    | TG^
99    | REC^
100   | FCAP^
101   | FCC^
102   | ECRIS^)
103   expr
```

```
104
105
106   | FPOS '[' a=expr b=expr ']' -> ^(FPOS $a $b)
107   | VE
108   | LC
109   | BC
110  // | boolExp
111   | set
112   | repete
113   | tantque
114   | alternative
115   | procedure
116   | procCall [false]
117   | retour
118 //  | STOP
119   ;
120
121
122 // Mathematics
123 expr   : sumExpr
124   ;
125
126 sumExpr : multExpr ((PLUS^|MOINS^) multExpr)*
127   ;
128
129 multExpr :
130   powExpr (( MULT^
131      | DIV^) powExpr)*
132 ;
133
134 powExpr :
135   atom (EXP^ atom)*
136   ;
137 atom :
138   '('! expr ')'!
139   | get
140   | INT
141   | (SIN^
142   | COS^) atom
143   | HASARD^ INT
144   | CAP
145   | procCall [true]
146     ;
147
148
149 //Boolean expr
150 boolExp :
151   boolTerm ((  OR^
152           | AND^) boolTerm)*
153 ;
154
155 boolTerm :
156   (expr (   '='^
157     | '<'^
158     | '>'^
```

```
159     | '<='^
160     | '=>'^) expr)
161   | TRUE
162   | FALSE
163 ;
164
165
166 // Vars - identificator
167 get :
168   GET^ ID { if( !(contexte.checkId($ID.text)) ){
169           Log.appendnl("Variable \"" + $ID.text + "\" non declaree");
170           valide=false;
171         }
172         }
173 ;
174
175 set :
176  DONNE^ '"'!ID expr {if( !(contexte.setId($ID.text, 0)) ){
177       Log.appendnl("Conflit : fonction ou variable \"" + $ID.text + "\" deja declaree");
178       valide=false;
179     }
180     }
181  ;
182
183
184
185 // Loops
186 loop_liste
187   :
188   '['^ liste_instructions ']'!
189 ;
190
191 repete  :
192  REPETE^ expr loop_liste
193 ;
194
195 alternative :
196  SI^ boolExp loop_liste loop_liste?
197 ;
198
199 tantque  :
200  TANTQUE^ boolExp loop_liste
201 ;
202
203
204
205 // Procedure - function
206 param_list
207   :
208   (':'ID {contexte.addParam($ID.text);} )+ -> ^(PARAM ID+)
209   ;
210
211
212 //param_list2 [int nbHe]
213 paramValue_list returns [int nbParam]
```

```
214 @init{
215 $nbParam = 0;
216 }
217   :
218   (expr {++$nbParam;})+
219   ;
220
221 instructions :
222   liste_instructions -> ^(INST liste_instructions)
223 ;
224
225 procedure
226 @init{
227 boolean test = false;
228 }
229 :
230  // Creation d'un context local a la procedure "ID",
231  // declaration des parametres et var locales
232  // puis recharge le contexte globale d'appel
233  POUR^ ID {test=contexte.createProcedure($ID.text);} param_list? instructions {if( test ){
         contexte.exitProcedure();}} FIN
234  {
235    // 1 - Traitement des erreurs
236    if( !(test) ){
237       Log.appendnl("Function \""+$ID.text+"\" or VAR already declared in this scope");
238       valide=test;
239    }
240
241  }
242 ;
243
244 procCall [boolean fromExpr]
245 :
246  ID^ (p=paramValue_list)? ';'! { if( !(contexte.checkProcCall($ID.text, $p.nbParam, $
         fromExpr)) )
247        {
248          valide=false;
249        }
250        }
251 ;
252
253
254 retour
255 @init{
256 boolean test = false;
257 }
258 :
259   RET^ expr
260   {
261   test=contexte.inProcedure();
262   if( !test){
263     valide=test;
264     Log.appendnl("Warning ! Return expr out of function");
265   }
266   else{
```

```
267     // Si oui -> la procedure est une fonction
268     contexte.setFunction();
269   }
270
271   }
272   ;
```

.2 LogoTree.g

Lexer et Parser - LogoTree.g

```
1  tree grammar LogoTree;
2  options {
3    tokenVocab = Logo;
4    ASTLabelType=CommonTree;
5  }
6  @header {
7    package logoparsing;
8    import logogui.Traceur;
9    import logogui.Log;
10   import java.lang.Math;
11 }
12 @members{
13   Traceur traceur;
14   public void initialize(java.awt.Graphics g) {
15     traceur = Traceur.getInstance();
16     traceur.setGraphics(g);
17   }
18     public void push(int index) {
19      ((CommonTreeNodeStream)input).push(index);
20   }
21   public void pop() {
22      ((CommonTreeNodeStream)input).pop();
23   }
24
25   LogoTable contexte;
26 }
27
28 prog [LogoTable scope]
29 @init{
30 contexte = scope;
31 } : ^(PROGRAMME (instruction)*)
32      {Log.appendnl("Programme principal");
33      System.out.println("--- Programme principal ---");}
34 ;
35
36 instruction :
37    ^(AV a = expr) {traceur.avance(a);}
38  | ^(TD a = expr) {traceur.td(a);}
39  | ^(TG a = expr) {traceur.tg(a);}
40  | ^(REC a = expr) {traceur.rec(a);}
41  | ^(FPOS a = expr b = expr) {traceur.fpos(a, b);}
42  | ^(FCAP a=expr) {traceur.fcap(a);}
43  | ^(FCC a=expr) {traceur.fcc(a);}
```

```
44  | ^(ECRIS a=expr) {traceur.ecris(a);}
45  | VE {traceur.ve();}
46  | LC {traceur.lc();}
47  | BC {traceur.bc();}
48  // | STOP {}
49  // | boolExp
50  | set
51
52  | repeat
53  | alternative
54  | tantque
55
56  | procedure
57  | procCall
58  | retour
59
60  ;
61
62
63  boolExp returns [boolean w] :
64    ^('=' a=expr b=expr) {$w = ($a.v == $b.v);}
65    | ^('<' a=expr b=expr) {$w = $a.v < $b.v;}
66    | ^('>' a=expr b=expr) {$w = $a.v > $b.v;}
67    | ^('>=' a=expr b=expr) {$w = $a.v >= $b.v;}
68    | ^('<=' a=expr b=expr) {$w = $a.v <= $b.v;}
69    | ^(AND x=boolExp y=boolExp) {$w =(($x.w)&&($y.w)) ;}
70    | ^(OR x=boolExp y=boolExp) {$w =(($x.w)||($y.w));}
71    | TRUE {$w = true;}
72    | FALSE {$w = false;}
73  ;
74
75
76  expr returns [double v] :
77  // Math
78  ^(PLUS a=expr b=expr) {$v = $a.v + $b.v;}
79  | ^(MOINS a=expr b=expr) {$v = $a.v - $b.v;}
80  | ^(MULT a=expr b=expr) {$v = $a.v * $b.v;}
81  | ^(DIV a=expr b=expr) {$v = $a.v / $b.v;}
82  | ^(EXP a=expr b=expr) {$v = Math.pow($a.v, $b.v);}
83
84  // Values - Vars
85  | ^(GET ID) {$v = contexte.getId($ID.text);}
86  | INT {$v = Double.parseDouble($INT.text);}
87  | h=hasard {$v = $h.value;}
88  | c=cap {$v=$c.value;}
89  | d=procCall {if(d!=null){$v=$d.value;}}
90  ;
91
92  // NOTE : J'AI CHANGE b=expr par w = boolExp ..... VOIR SI OK
93  set :
94    ^(DONNE ID b=expr){contexte.setId($ID.text, $b.v);}
95  ;
96
97
98  // Autres methodes
```

```
99  hasard returns [double value] :
100   ^(HASARD INT)
101   {
102     double max = Double.parseDouble($INT.text);
103     $value = (Math.random() * (max));
104     //Log.appendnl("Hasard : "+value);
105   }
106 ;
107
108
109 cap returns [double value] :
110   CAP {
111     $value = traceur.cap();
112     //Log.appendnl("cap : "+value);
113   }
114 ;
115
116
117
118 // LOOPS
119 liste_instructions
120   :
121   ^('[' (instruction)+ )
122   | ^(INST (instruction)+ )
123   ;
124
125 repeat
126 @init{
127 int mark_list = 0;
128 }
129 : ^(REPETE a=expr {mark_list = input.mark();} . )
130   {
131   for (int i = 0; i < $a.v ; i++) {
132    ((CommonTreeNodeStream)input).push(mark_list);
133    liste_instructions();
134    ((CommonTreeNodeStream)input).pop();
135   }
136   }
137   ;
138
139 alternative
140 @init{
141 int mark_list = 0;
142 int mark_list2 = -1;
143 }
144 : ^(SI x=boolExp ({mark_list = input.mark();} .) ({mark_list2 = input.mark();} .)?)
145   {
146   //Log.appendnl("SI : mark_list = "+mark_list+" mark_liste2 = "+mark_list2);
147   // Parse de l'arbre
148   // On memorise l'empalcement de list
149   // on memorise l'emplacement de liste2 si existe
150     if($x.w == true ){
151        ((CommonTreeNodeStream)input).push(mark_list);
152        // liste_instruction() va chercher dans la pile l'indice de la branche a executer
153        // donc on push son indice avant
```

```
154       liste_instructions();
155       // a la fin, liste_instructions() push l'indice de la prochaine instruction a
              executer
156       // on pop pour retrouver l'indice normal
157       ((CommonTreeNodeStream)input).pop();
158     }
159     else if (mark_list2 != -1){
160       ((CommonTreeNodeStream)input).push(mark_list2);
161       liste_instructions();
162       ((CommonTreeNodeStream)input).pop();
163     }
164   }
165 ;
166
167 tantque
168 @init{
169 int mark_x = 0;
170 int mark_list = 0;
171 boolean x=false;
172 }
173 : ^(TANTQUE ({mark_x = input.mark();} .) ({mark_list = input.mark();} .) )
174   {
175     // on met l'indice de X dans la pile
176     ((CommonTreeNodeStream)input).push(mark_x);
177     // boolExp() va chercher dans la pile l'instruction a evaluer
178     x=boolExp();
179         ((CommonTreeNodeStream)input).pop();
180
181     while(x == true ){
182       // Push l'indice de la liste d'instruction
183       ((CommonTreeNodeStream)input).push(mark_list);
184       liste_instructions();
185       ((CommonTreeNodeStream)input).pop();
186
187       // On recupere la nouvelle valeur de X
188       ((CommonTreeNodeStream)input).push(mark_x);
189       x=boolExp();
190             ((CommonTreeNodeStream)input).pop();
191     }
192   }
193   ;
194
195
196 // PROCEDURES
197 param_list
198 : ^(PARAM (ID)+)
199 ;
200
201 paramValue_list
202 : (a=expr {contexte.addParam($a.v);})+
203 ;
204
205 instructions :
206   ^(INST (.)+ )
207 ;
```

```
208
209 procedure
210 @init{
211 int mark_instructions = -1;
212 }
213 :
214   ^(POUR ID (.)? {mark_instructions = input.mark();} instructions FIN)
215   {
216    // 1- Recupere le contexte local declare en 1ere passe
217    contexte.getProcedure($ID.text);
218
219    // 2 - Place l'index de la procedure dans le contexte local
220    if( mark_instructions != -1){
221     contexte.setProcIndex(mark_instructions);
222     //Log.appendnl("procedure : declaration de la fonction \""+$ID.text+"\" mark_instruction
          = "+mark_instructions);
223    }else {
224     //Log.appendnl("procedure : ERROR ? declaration de la fonction \""+$ID.text+"\" pas d'
          instruction");
225    }
226    // 3 - Retour au contexte (n+1)
227    contexte.getPreviousProcedureReference();
228   }
229 ;
230
231
232 procCall returns [Double value]
233 @init{
234 int mark_instructions = -1;
235 int mark_param_list = -1;
236 int i=0;
237 }:
238  ^(ID ({mark_param_list = input.mark();} (.)*) )
239  {
240    // 1 - Recupere le contexte local declare en 1ere passe
241    if( !contexte.getLocalContext($ID.text) ){
242     return null;
243    }
244
245    // 2 - initialisation des parametres
246    if(mark_param_list != -1){
247     ((CommonTreeNodeStream)input).push(mark_param_list);
248   paramValue_list();
249   ((CommonTreeNodeStream)input).pop();
250   contexte.initProc();
251   //Log.appendnl("procCall : \""+$ID.text+"\" initialisation param OK ");
252    }else {Log.appendnl("procCall : \""+$ID.text+"\" pas de parametres ");}
253
254
255    // 3 - Recupere l'index des instructions
256    mark_instructions = contexte.getProcIndex();
257    //mark_instructions = contexte.getId($ID.text).intValue();
258    //Log.appendnl("procCall : Appelle de \""+$ID.text+"\" mark_liste = "+mark_instructions);
259
260
```

```
261     // 4 - Execution des instuctions
262     ((CommonTreeNodeStream)input).push(mark_instructions);
263     liste_instructions();
264     ((CommonTreeNodeStream)input).pop();
265
266
267     // 5 - Si procedure = fonction => retourne une valeur
268     if(contexte.isFunction()){
269      $value = contexte.getFunctionValue();
270     }
271     else {$value = null;}
272
273
274     // 5 - Retour au contexte (n+1)
275     contexte.exitProcedure();
276  }
277 ;
278
279
280
281 retour :
282 ^(RET a=expr) {contexte.setFunctionValue($a.v);}
283   ;
```