Tanguy ROZIER – Mickaël BARROUX

Rapport NF11

Génération d'analyseur lexical et syntaxique

Table des matières

INTRODUCTION 3
GRAMMAIRE 3
Lexer 3
Parser 3
Instructions de base 4
Expressions arithmétiques 4
Expressions booléennes 4
Variables 4
Tree-walker 5
STRUCTURE 5
L’alternative 5
Lexer 6
Parser 6
Tree-walker 6
La boucle TANT QUE 7
Lexer 7
Parser 7
Tree-walker 7
La boucle REPETE 8
Lexer 8
Parser 8
Tree-walker 8
Les procédures et fonctions 9
Déclaration d’une procédure ou fonction 9
Lexer 9
Parser 9
Tree-walker 9
Appel d’une procédure ou d’une fonction 10
Parser 10
Tree-walker 10
Récursivité 11
ELEMENTS NON TRAITES 11
DIFFICULTES RENCONTRES 12
EXEMPLES D’EXECUTIONS 12
CONCLUSION 14
ANNEXES 14
Lexer et parser 14
Tree-walker 17
Traceur 22
Procedure 24
LogoContext 25

Introduction

Le Logo est un langage de programmation élémentaire qui permet de diriger une tortue et de tracer des figures géométriques.

Comme tout langage de programmation, Logo implémente les structures algorithmiques telles que les conditions, les boucles, les procédures et fonctions, la récursivité qui lui permettent de dessiner des figures de plus en plus complexes et évoluées.

Le but du projet est de créer un interpréteur de ce langage, grâce à l'outil *AntLR*, qui génère le parser Java à partir des règles de la grammaire.

Ce rapport présente la démarche suivie lors de la réalisation de cet interpréteur, en commençant par le découpage de la grammaire, puis en décrivant les différentes structures et caractéristiques de cette grammaire. Enfin, nous aborderons les problèmes rencontrés lors du projet puis présenterons différents résultats d'exécution de l'interpréteur Logo réalisé.

Grammaire

Le projet est réparti en 2 fichiers de grammaires AntLR (2 passes) : *Logo.g* qui correspond au lexer et au parser et *LogoTree.g* qui correspond au tree-walker. *Logo.g* se charge des analyses lexicale et syntaxique tandis que *LogoTree.g* se charge de l'analyse sémantique et de l'exécution des instructions.

Lexer

Le lexer répertorie la déclaration des mots-clés du langage Logo, qui seront utilisés pour générer le flux de tokens utilisé par le parser.

```
tokens {
  PROGRAMME;
  AV = 'AV';
  TD = 'TD';
  TG = 'TG';
  REC = 'REC';
  FPOS = 'FPOS';
  FCAP = 'FCAP';
  ...
}
REAL : ('0'..'9')+('.'('0'..'9')+)? ;
WS  :   (' '|'\t'|('\r'? '\n'))+ { skip(); } ;
ID : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z'|'0'..'9'|'_')*;
```

Les tokens *REAL* représentent les nombres flottants et les tokens *ID* les identificateurs (noms de fonction, de paramètres, etc...) : ils doivent donc commencer obligatoirement par une lettre, suivi d'une suite de lettres, de chiffres ou du caractère '_'.

Parser

Le parser crée l'arbre syntaxique abstrait (AST) à partir du flux de tokens généré par le lexer. Les différentes règles permettent d'identifier les différentes structures du langage, décrites un peu plus loin. L'axiome de la grammaire est la règle *programme* qui est constituée d'une liste d'instructions :

```
programme [LogoContext scope]
```

```
@init{
  context = scope;
  context.init();
} : liste_instructions -> ^(PROGRAMME liste_instructions);
```

Instructions de base

Le langage dispose de commandes de base, permettant à la tortue de se déplacer dans son environnement.

```
instruction :
  (( AV^ | TD^ | TG^ | REC^ | FCAP^ | FCC^) expr) |
  (VE^ | MT^ | CT^ | LC^ | BC^) |
  (FPOS^ '['! expr expr ']'!) |
  ...
;
```

La règle *instruction* référence toutes les instructions possibles du langage Logo, dont les instructions de base, mais aussi les structures spécifiques détaillées plus loin (conditions, boucles...).

Expressions arithmétiques

Il est nécessaire d'implémenter les expressions arithmétiques de base, afin d'additionner, soustraire, multiplier ou diviser des éléments *atom*, c'est-à-dire des nombres flottants, des variables déclarées ou bien des variables spécifiques au langage (*HASARD*, *CAP*). La priorité des opérateurs multiplier et diviser sur les opérateurs additionner et soustraire est implémentée.

```
expr: mexpr ((PLUS^ | MOINS^) mexpr)*;
mexpr: atom ((MULT^ | DIV^) atom)*;
atom: REAL | CAP | hasardExpr | LOOP{ setVar("LOOP",0.0); }
      | ':'!ID { if($ID.getText() != "LOOP") getVar($ID.getText()); }
      |  LPAREN! expr RPAREN!
;
```

Expressions booléennes

Les expressions logiques sont implémentées de la même façon que les expressions arithmétiques : elles permettent de créer des conditions booléennes nécessaires aux structures Logo telles que l'alternative, la répétition, etc...

```
boolnexpr: expr ((EGAL^ | LESSTHAN^ | MORETHAN^ | LESSOREQ^ | MOREOREQ^) expr);
```

Variables

La déclaration et l'affectation d'une variable sont effectuées grâce au mot-clé *DONNE*, qui stocke le nom et la valeur de la variable dans une table.

```
instruction :
  ...
  (DONNE^ '"'! ID a=expr) { setVar($ID.getText(),0.0); } |
  ...
;
```

Les variables sont stockées dans une table de hashage (*HashMap*) qui associe le nom de la variable à sa valeur. La gestion des variables locales à un bloc d'instructions (mot-clé *LOCALE*) impose l'utilisation d'une pile de contextes, représentée par la classe *LogoContext*. Cette classe permet d'empiler les tables des variables dans lesquelles sont stockées les valeurs des variables du contexte courant.

```
// ajoute une nouvelle variable name avec comme valeur value au contexte courant
public void setVar(String name, double value) {
    Double p = new Double(value);
    HashMap<String,Double> h = (HashMap<String,Double>)context.peek();
    h.put(name, p);
}
```

Lors de la lecture d'une variable (comme élément *atom* de la règle *expr*), la pile de contextes est parcourue depuis son sommet jusqu'à trouver la valeur de cette variable. Si elle n'est pas trouvée, on va la chercher dans le contexte de niveau inférieur, et ainsi de suite...

```
// récupère la valeur de la variable name
public double getVar(String name) {
    Double value = null;
    Stack pile = context.getPile();
    HashMap<String,Double> h;
    for(int i=pile.size()-1; i>=0; i--){
        h = (HashMap<String,Double>)pile.get(i);
        value = h.get(name);
        if (value!=null) {
            if(i==pile.size()-1 && value.doubleValue() == -1.0){
                Log.appendnl("variable locale "+name+" non initialisée");
                return 0.0; // valeur par défaut
            }
            return value.doubleValue();
        }
    }
    // test dans les paramètres de fonctions
    if(context.getVarInProcedures(name)) return 0.0;
    if (value==null){
        if(name == "loop") Log.appendnl("instruction LOOP utilisée en dehors d'un
REPETE");
        else Log.appendnl("undefined variable "+name);
        valide=false;
    }
    return 0.0;
}
```

Tree-walker

Le tree-walker parcourt l'arbre AST généré par le parser et exécute les instructions correspondantes en faisant appel à la classe *Traceur*.

Ainsi l'interprétation d'une instruction du langage Logo s'écrit de la façon suivante :

```
instruction
@init{
   int mark_before_true = 0; int mark_before_false = 0; int marker_fin = 0;
} :
    ^(AV a = expr) {traceur.avance(a);}
  | ^(TD a = expr) {traceur.td(a);}
  | ^(TG a = expr) {traceur.tg(a);}
  ...
;
```

Structure

L'alternative

Si *prédicat* est vrai, exécute les instructions contenues dans *liste1* ou sinon celles de *liste2* (optionnelle) :

```
SI prédicat [liste1] [liste2]
```

Lexer

```
IF = 'SI';
```

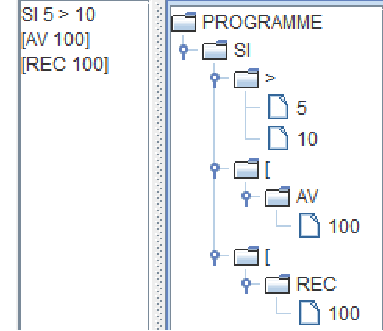
Parser

Pour générer l'AST correspondant, on déclare le token *IF* comme racine du sous-arbre contenant la condition (*boolnexpr*), le premier bloc et éventuellement le second bloc d'instructions (*block*).

Un bloc d'instructions est un sous-arbre ayant pour racine le caractère '[' et pour fils les instructions à exécuter. Quand on entre dans un bloc, on crée un nouveau contexte que l'on empile et que l'on dépile lors de la sortie de ce bloc.

```
instruction :
   ...
   (IF^ boolnexpr block block?) |
   ...
;
block
@init{
       HashMap<String,Double> h =
       new HashMap<String,Double>();
       context.push(h);
}
@after{
       context.pop();
} : '['^ liste_instructions ']'!;
```

Exemple :



Tree-walker

A partir de l'AST généré par le parser, le tree-walker récupère la condition dans *e*, marque le début du premier bloc d'instructions et du second s'il existe. Ensuite il crée un nouveau contexte de bloc puis si la condition *e* est vraie, exécute les instructions de la première liste, et celles de la seconde si elle est fausse (et si la seconde liste existe).

```
instruction
@init{
   int mark_before_true = 0;  int mark_before_false = 0;  int marker_fin = 0;
} :
   ...
   ^(IF e=boolnexpr {mark_before_true=input.mark();} . ({mark_before_false =
   input.mark();} else_list=.)? )
   {
       HashMap<String,Double> h = new HashMap<String,Double>();
       if (e){
              context.push(h);
              push(mark_before_true);
              list_instructions();
              pop();
              context.pop();
       }
       else if (else_list != null) {
              context.push(h);
              push(mark_before_false);
              list_instructions();
              pop();
              context.pop();
       }
   }
   ...
```

```
;
list_instructions: ^('[' instruction+);
```

La boucle TANT QUE

Répète les instructions contenues dans *liste* tant que *exp* est vraie.

```
TANQUE exp [liste]
```

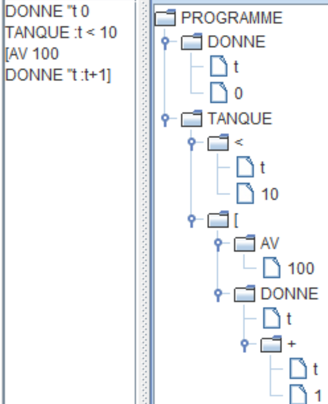
Lexer

```
WHILE = 'TANQUE';
```

Parser

L'AST généré correspondant à la structure *TANQUE* a pour racine le token *WHILE* et a comme fils la condition (*boolnexpr*) et le bloc d'instructions à exécuter (*block*).

```
instruction :
  ...
  (WHILE^ boolnexpr block) |
  ...
;
```



Tree-walker

Le tree-walker récupère le sous-arbre ayant pour racine le token *WHILE* et marque le début de la condition qui sera réévaluée à chaque passage dans la boucle. Il marque ensuite le début des instructions à exécuter. Tant que la condition est évaluée comme vraie, on exécute les instructions à partir de la marque (*mark_before_false*).

```
instruction
@init{
  int mark_before_true = 0;  int mark_before_false = 0;  int marker_fin = 0;
} :
  ...
  ^(WHILE {mark_before_true=input.mark();} condition=. {mark_before_false =
  input.mark();} suite=.)
  {
        push(mark_before_true+1); // +1 sinon prend le token <DOWN> après le TANQUE
        boolean booleen = boolnexpr();
        pop();
        while (booleen) {
              // Répétition des instructions de la boucle
              push(mark_before_false);
              list_instructions();
              pop();
              // Test de condition d'exécution de la boucle suivante
              push(mark_before_true+1) ;
              booleen = boolnexpr();
              pop();
        }
  }
  ...
;
```

La boucle REPETE

Répète *n* fois les instructions contenues dans *liste*

```
REPETE n [liste]
```

Lexer

```
REPETE = 'REPETE';
```

Parser

Le sous-arbre AST généré a comme racine le token *REPETE* et a comme fils le nombre de répétitions et le bloc d'instructions à exécuter.

```
instruction :
  ...
  (REPETE^ expr block) |
  ...
;
```

Tree-walker

Le tree-walker récupère le nombre de fois que l'on doit exécuter les instructions dans *a* et marque le début du bloc d'instructions (*mark_before_true*). Il crée ensuite un nouveau contexte pour le bloc d'instructions et tant que le nombre d'exécutions est supérieur strictement à 0, il exécute les instructions marquées et décrémente le nombre d'exécutions restantes.

Il crée également au début de la règle une variable *cpt* qu'il initialise à 0 et à chaque exécution des instructions, il incrémente cette variable. Il crée une nouvelle variable de contexte avec comme nom « LOOP » et lui affecte la valeur de *cpt*. C'est nécessaire afin de pouvoir récupérer avec l'instruction Logo *LOOP* le nombre de fois que l'instruction *REPETE* a été exécutée.

```
instruction
@init{
  int mark_before_true = 0;  int mark_before_false = 0;  int marker_fin = 0;
} :
  ...
  ^(REPETE a=expr {mark_before_true = input.mark();} .)
  {
       int cpt = (int)a;
       int nb_exec = 0;
       HashMap<String,Double> h = new HashMap<String,Double>() ;
       context.push(h);
       while(cpt > 0){
              setVar("LOOP", nb_exec); // nombre de fois que REPETE a été exécuté
              push(mark_before_true);
              list_instructions();
              pop();
              cpt--;
              nb_exec++;
       }
       context.pop();
  }
  ...
;
```

Les procédures et fonctions

La gestion des procédures et des fonctions est identique, la seule différence est l'utilisation de l'instruction logo *RENDS* qui permet de retourner une valeur à la fonction appelante sans exécuter la suite des instructions (une procédure ne renvoie quant à elle pas de valeur).

Déclaration d'une procédure ou fonction

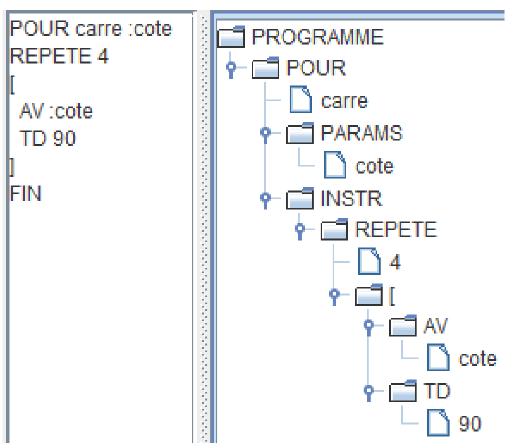
Lexer

```
POUR = 'POUR';
```

Parser

On définit un sous-arbre ayant pour racine le token *POUR*, et pour fils :

- ✓ le nom de la procédure
- ✓ un sous-arbre avec pour racine le token imaginaire *PARAMS* ayant pour fils la liste des noms des paramètres à passer à la procédure lors de son appel
- ✓ un sous-arbre avec pour racine le token imaginaire *INSTR* ayant pour fils le bloc d'instruction à exécuter




```
instruction :
   ...
   (POUR^ ID { addProcedure($ID.getText()); } liste_params blockPour) |
   ...
;
blockPour
@init{
        HashMap<String,Double> h = new HashMap<String,Double>();
        context.push(h);
}
@after{
        context.pop();
} : liste_instructions 'FIN' -> ^(INSTR liste_instructions);

liste_params : liste_parametres -> ^(PARAMS liste_parametres?);
liste_parametres : (parametre_fonction)*;
parametre_fonction : ':'! ID
        {tmp.addNomParam($ID.getText()); empileNomParam( "PARAM_"+$ID.getText());};
```

Tree-walker

Le tree-walker marque les positions du code (début et fin), et le début de la liste des noms des paramètres. Il instancie la classe *Procedure* en y stockant : nom, marques de début et de fin de code, liste des noms des paramètres (vecteur). La procédure est ensuite ajoutée à la table des procédures de *LogoContext*.

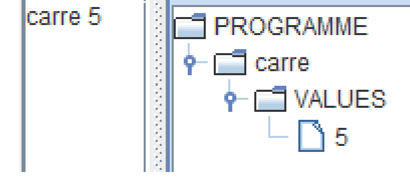
```
instruction :
@init{
  int mark_before_true = 0;
```

```
  int mark_before_false = 0;
  int marker_fin = 0;
} :
  ...
  ^(POUR ID {mark_before_true=input.mark();} . ({mark_before_false = input.mark();}
  code=.)? {marker_fin = input.mark();} )
  {
       Procedure p = new Procedure($ID.getText());
       p.setFinBloc(marker_fin-1);
       if(code==null){ // pas de paramètre
              p.setDebut(mark_before_true);
              tmp=p;
       }else{ // paramètres
              p.setDebut(mark_before_false);
              // évite de chercher de nouveau la procédure dans les contextes
              // quand on récupère les noms des paramètres...
              tmp=p;
              // récupère les noms des paramètres
              push(mark_before_true);
              list_nomparams();
              pop();
       }
       context.addProcedure(p);
  }
  ...
;
```

Appel d'une procédure ou d'une fonction

Parser

Le parser, lorsqu'il rencontre l'identifiant d'une procédure, construit un sous-arbre ayant pour racine le nom de la procédure et pour fils un sous-arbre ayant pour racine un token imaginaire *VALUES* qui a pour fils la liste des valeurs des paramètres passées à la procédure.




```
instruction :
  ...
  appelProc2 |
  ...
;
appelProc2 : (ID^ valParams) { appelProc($ID.getText()); };
valParams : ((expr {empileValParam(0.0);})*) -> ^(VALUES (expr)*);
```

Tree-walker

Le tree-walker, lorsqu'il trouve un identificateur racine d'un sous-arbre de l'AST généré par le parser, marque le début de la liste des valeurs des paramètres de la procédure.

Il récupère ces valeurs, qu'il stocke dans un vecteur, afin de faire plus tard la correspondance avec les noms des paramètres de la procédure.

Il empile le contexte avec ces valeurs de variables, puis saute jusqu'au code du bloc d'instructions à exécuter (code défini dans la déclaration de la procédure).

Lorsqu'il a fini d'exécuter le bloc d'instructions, il dépile le contexte de la fonction en stockant dans une variable du contexte père la valeur de retour (nommée « RENDS »).

```
instruction :
  ...
```

```
  appelProcedure |
  ...
;
appelProcedure returns [double valeurRetour=0]
@init{  int mark_before_true = 0; }
:
  ^(ID {mark_before_true = input.mark();} .)
  {
       paramsValues = new Vector<Double>();
       push(mark_before_true+1);
       list_valparams2();
       pop();
       Procedure p = context.getProcedure($ID.getText());
       tmp = p;
       int pos = p.getDebut();
       push(pos);
       // correspondance NOMS/VALEURS des paramètres
       HashMap<String,Double> h = new HashMap<String,Double>();
       Vector<String> noms = p.getNomsParams();
       for(int i=0; i<noms.size(); i++){
              String nom = noms.get(i);
              Double val = paramsValues.get(i);
              h.put(nom,val);
       }
       context.push(h);
       list_instructions_procedure(); // exécution de la procédure
       context.pop();
       pop();
       // valeur de retour de la procédure appelée si elle existe / 0 sinon
       valeurRetour = getReturnValue();
 };
list_instructions_procedure: ^(INSTR instruction+);

// récupère la valeur de retour de la fonction appelée
public double getReturnValue() {
       Double p;
       HashMap<String,Double> h = (HashMap<String,Double>)context.getPile().get(
       context.getPile().size()-1); // hashmap des variables du contexte appelant
       p = h.get("RENDS"); // nom fixé pour la variable de retour
       if(p==null) return 0.0;
       else return p.doubleValue();
}
```

Récursivité

La récursivité est implicitement gérée par le mécanisme de pile de contextes des variables.

A noter que la machine Java de SUN limite la taille de la pile. Il est donc nécessaire de passer les arguments suivants à l'exécution :

```
-XX:NewSize=128m -XX:MaxNewSize=128m -XX:SurvivorRatio=8 -Xms512m -Xmx512m
```

Ceci afin d'agrandir la taille de la pile (sinon on obtient vite une StackOverflowError).

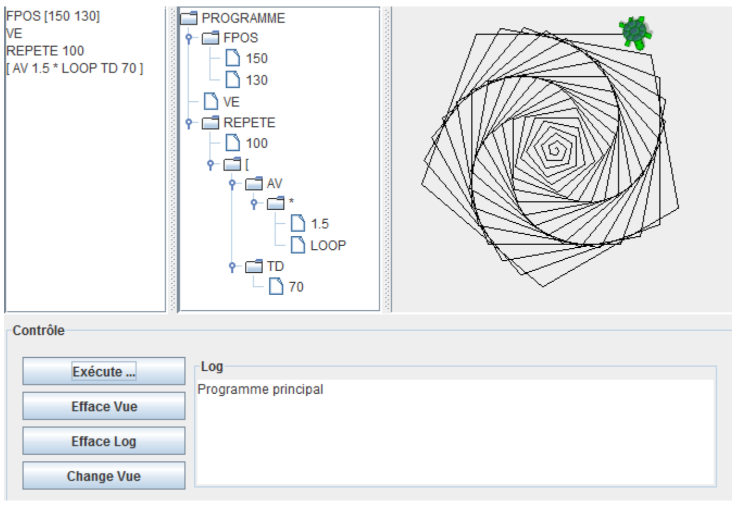
Eléments non traités

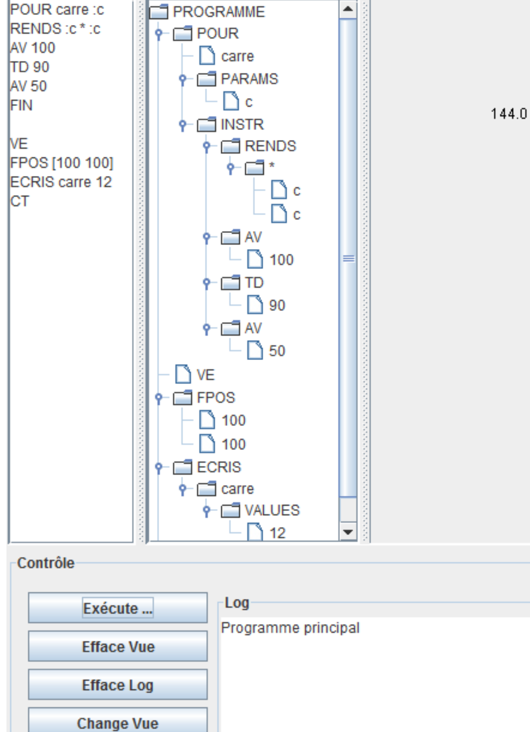
- ✓ La primitive *STOP* n'a pas été implémentée faute de temps. En revanche, lors de l'utilisation de la commande Logo *RENDS* dans une fonction, la suite des instructions n'est pas exécutée. On pourrait s'en inspirer pour réaliser la primitive *STOP*.
- ✓ Les expressions booléennes ne permettent pas d'employer les opérateurs binaires *AND* et *OR*.

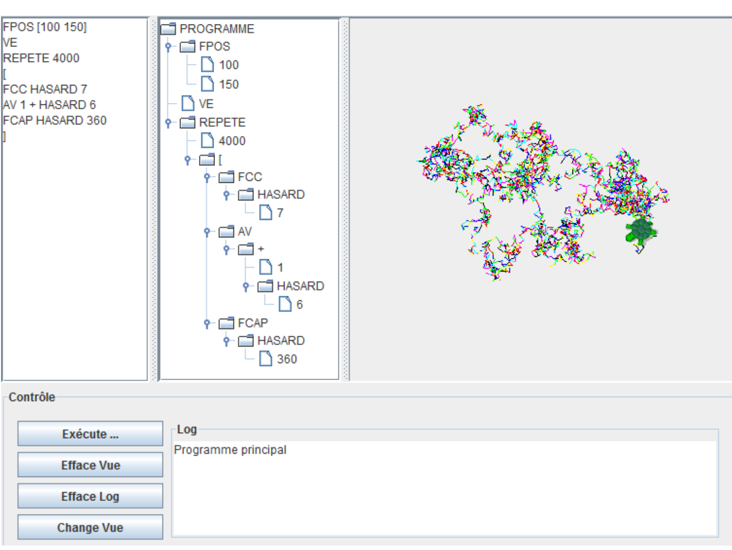
Difficultés rencontrés

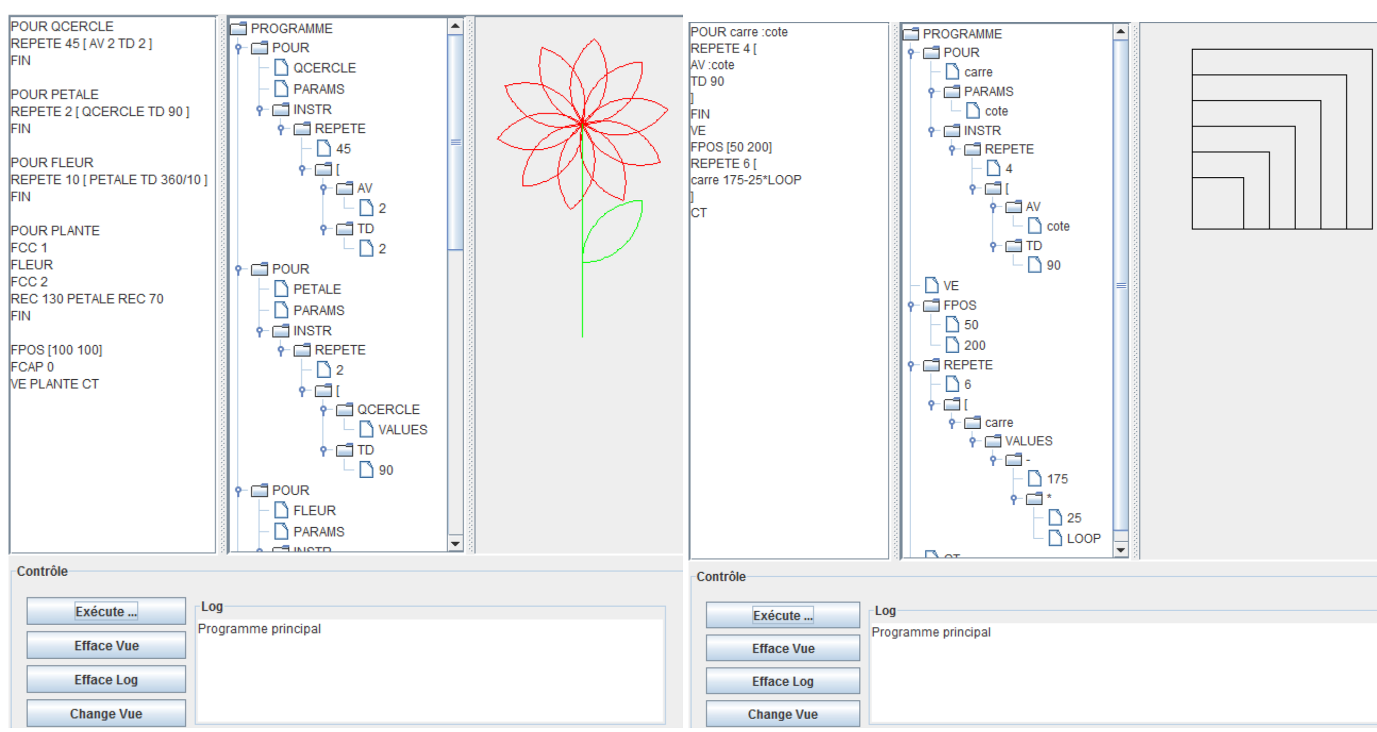
- ✓ La syntaxe *AntLR* a nécessité un certain temps d'apprentissage et d'adaptation
- ✓ Les problèmes liés aux tokens imaginaires *<UP>* et *<DOWN>* dans *AntLR* lors du parsing
- ✓ La nécessité de créer des tokens imaginaires pour structurer l'AST généré par le parser

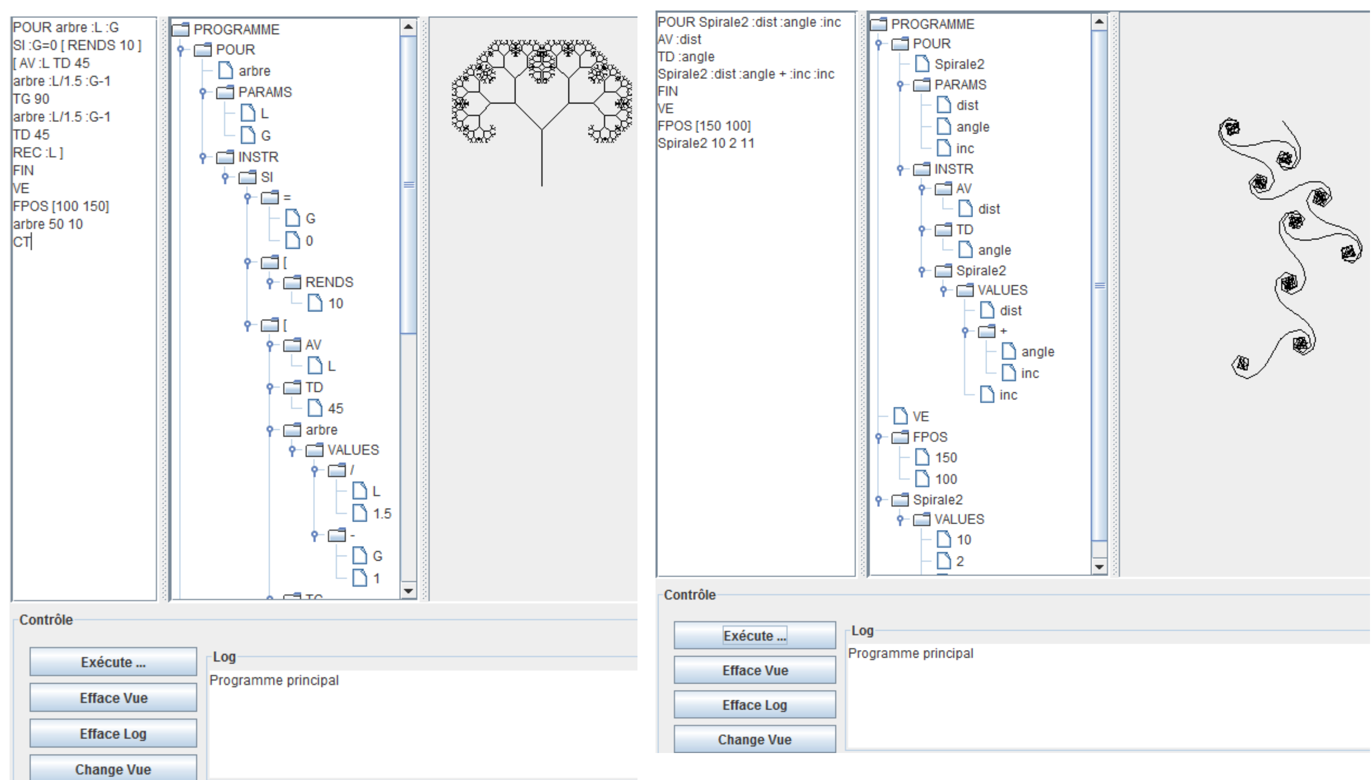
Exemples d'exécutions








POUR QCERCLE
REPETE 45 [AV 2 TD 2]
FIN
POUR PETALE
REPETE 2 [QCERCLE TD 90]
FIN
POUR FLEUR
REPETE 10 [PETALE TD 360/10]
FIN
POUR PLANTE
FCC 1
FLEUR
FCC 2
REC 130 PETALE REC 70
FIN
FPOS [100 100]
FCAP 0
VE PLANTE CT
PROGRAMME
Contrôle
Exécute ...
Efface Vue
Efface Log
Change Vue
Log
Programme principal
POUR carre :cote
REPETE 4 [
AV :cote
TD 90
]
FIN
VE
FPOS [50 200]
REPETE 6 [
carre 175-25*LOOP
]
CT
PROGRAMME
Contrôle
Exécute ...
Efface Vue
Efface Log
Change Vue
Log
Programme principal


POUR arbre :L :G
SI :G=0 [RENDS 10]
[AV :L TD 45
arbre :L/1.5 :G-1
TG 90
arbre :L/1.5 :G-1
TD 45
REC :L]
FIN
VE
FPOS [100 150]
arbre 50 10
CT
PROGRAMME
Contrôle
Exécute ...
Efface Vue
Efface Log
Change Vue
Log
Programme principal
POUR Spirale2 :dist :angle :inc
AV :dist
TD :angle
Spirale2 :dist :angle + :inc :inc
FIN
VE
FPOS [150 100]
Spirale2 10 2 11
PROGRAMME
Contrôle
Exécute ...
Efface Vue
Efface Log
Change Vue
Log
Programme principal

Conclusion

Une fois *AntLR* pris en main, nous avons implémenté une à une les différentes structures du langage Logo, en étoffant petit à petit notre grammaire et en ajoutant les actions associées.

Ce projet nous a paru intéressant dans la mesure où il nous a permis d'appréhender la façon dont les langages de programmation sont écrits et interprétés au plus bas niveau, c'est-à-dire par exemple dans un compilateur, qui ne fait que lire des flux de caractères qu'il doit ensuite interpréter !

Annexes

Lexer et parser

```
grammar Logo;
options {
  output = AST;
}
tokens {
  PROGRAMME;
  AV = 'AV';
  TD = 'TD';
  TG = 'TG';
  REC = 'REC';
  FPOS = 'FPOS';
  FCAP = 'FCAP';
  VE = 'VE';
  MT = 'MT';
  CT = 'CT';
  LC = 'LC';
  BC = 'BC';
  FCC = 'FCC';
  CAP = 'CAP';
  HASARD = 'HASARD';
  PLUS = '+';
  MOINS = '-';
  MULT = '*';
  DIV = '/';
  EGAL = '=';
  LPAREN = '(';
  RPAREN = ')';
  DONNE = 'DONNE';
  LOCALE = 'LOCALE';
  LESSTHAN = '<';
  MORETHAN = '>';
  LESSOREQ = '<=';
  MOREOREQ = '>=';
  IF = 'SI';
  BLOC = 'BLOC';
  REPETE = 'REPETE';
  LOOP = 'LOOP';
  WHILE = 'TANQUE';
  POUR = 'POUR';
  RETURN = 'RENDS';
  ECRIS = 'ECRIS';
  FIN = 'FIN';
  PARAMS;
  INSTR;
  APPEL;
  VALUES;
  }
@lexer::header {
  package logoparsing;
```

```
}
@header {
  package logoparsing;
  import logogui.LogoContext;
  import logogui.Procedure;
  import java.util.Map;
  import java.util.HashMap;
  import logogui.Log;
  import java.util.Collection;
  import java.util.Iterator;
  import java.util.Vector;
}
@members{
  boolean valide = true;
  LogoContext context;

  private Vector<String> paramsNoms;
  private Vector<Double> paramsValues;
  private Procedure tmp;

    public boolean getValide(){
        return valide;
    }

    // récupère la valeur de la variable name
    public double getVar(String name) {
       Double value = null;
       Stack pile = context.getPile();
       HashMap<String,Double> h;

       for(int i=pile.size()-1; i>=0; i--){
              h = (HashMap<String,Double>)pile.get(i);

              value = h.get(name);
               if ( value!=null ) {
                    if(i==pile.size()-1 && value.doubleValue() == -1.0){
                           Log.appendnl("variable locale "+name+" non initialisée");
                           return 0.0; // valeur par défaut
                    }
                    return value.doubleValue();
               }
       }
       // test dans les paramètres de fonctions
       if(context.getVarInProcedures(name)) return 0.0;
        if ( value==null ) {
              if(name == "loop") Log.appendnl("instruction LOOP utilisée en dehors
d'un REPETE");
              else Log.appendnl("undefined variable "+name);
              valide=false;
        }
        return 0.0;
    }

    // ajoute une nouvelle variable name avec comme valeur value au contexte
courant
    public void setVar(String name, double value) {
        Double p = new Double(value);
        HashMap<String,Double> h = (HashMap<String,Double>)context.peek();
              h.put(name, p);
    }

    // ajoute un paramètre d'appel qui a pour valeur value
    public void empileValParam(double value){
       Double p = new Double(value);
              paramsValues.add(p);
    }

    // ajoute un nom de paramètre lors de la déclaration d'une procédure
```

```
    public void empileNomParam(String value){
       paramsNoms.add(value);
    }

    // vérifie que le nb de paramètres passés lors de l'appel d'une procédure
    // est égal au nb de paramètres déterminé dans la déclaration de la fonction
    public void ariteProcedure(int nbParamsDeclared, int nbParamsPassed){
       if(nbParamsPassed != nbParamsDeclared){
             Log.appendnl("wrong number of parameters ("+nbParamsPassed+") :
expected " + nbParamsDeclared);
             valide = false;
       }
    }

    // renvoie vrai si name est un paramètre de la fonction courante (pour empêcher
une variable locale de s'appeler pareil)
    public boolean alreadyNomParam(String name){
       boolean bool = false;
       bool = paramsNoms.contains("PARAM_"+name);
         if (bool){
             Log.appendnl("Impossible de déclarer une variable locale qui a le meme
nom qu'un paramètre : "+name);
             valide = false;
             return true;
         }else{
             return false;
         }
    }

    // vérifie l'arité de la procédure et si elle existe quand on l'appelle
    public void appelProc(String name){
       Procedure p = context.getProcedure(name);
       if(p == null){
             valide = false;
             Log.appendnl("Procédure "+name+" inconnue !");
       }else{
             ariteProcedure(p.getNbParams(),paramsValues.size());
       }
       paramsValues.clear();
    }

    // ajoute la procédure name au contexte
    public void addProcedure(String name){
       Procedure p = new Procedure(name);
       tmp = p;
       context.addProcedure(p);
       paramsNoms.clear();
    }
}

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

programme [LogoContext scope]
@init{
  context = scope;
  context.init();
  paramsNoms = new Vector<String>();
  paramsValues = new Vector<Double>();
}
:
  liste_instructions -> ^(PROGRAMME liste_instructions)
;

liste_instructions : (instruction)+ ;

instruction :
```

```
  (( AV^ | TD^ | TG^ | REC^ | FCAP^ | FCC^) expr) |
  (VE^ | MT^ | CT^ | LC^ | BC^) |
  (FPOS^ '['! expr expr ']'!) |
  (DONNE^ '"'! ID a=expr) { if($ID.getText() != "LOOP" && $ID.getText() != "RENDS")
setVar($ID.getText(),0.0); } |
  (ECRIS^ exprOuAppel) |
  (LOCALE^ '"'! ID) { if($ID.getText() != "LOOP" && $ID.getText() != "RENDS" &&
!alreadyNomParam($ID.getText())) setVar($ID.getText(),-1.0); } |
  (IF^ boolnexpr block block?) |
  (REPETE^ expr block) |
  (WHILE^ boolnexpr block) |
  (POUR^ ID { addProcedure($ID.getText()); } liste_params blockPour) |
  appelProc2 |
  (RETURN^ expr)
;

expr: mexpr ((PLUS^ | MOINS^) mexpr)*;
mexpr: atom ((MULT^ | DIV^) atom)*;
atom: REAL | CAP | hasardExpr | LOOP{ setVar("LOOP",0.0); }
      | ':'!ID { if($ID.getText() != "LOOP") getVar($ID.getText()); }
      |  LPAREN! expr RPAREN!
;

boolnexpr: expr ((EGAL^ | LESSTHAN^ | MORETHAN^ | LESSOREQ^ | MOREOREQ^) expr);

hasardExpr : HASARD^ expr;

block
@init{
      HashMap<String,Double> h = new HashMap<String,Double>();
      context.push(h);
}
@after{
      context.pop();
} : '['^ liste_instructions ']'!;

blockPour
@init{
      HashMap<String,Double> h = new HashMap<String,Double>();
      context.push(h);
}
@after{
      context.pop();
} : liste_instructions 'FIN' -> ^(INSTR liste_instructions);

liste_params : liste_parametres -> ^(PARAMS liste_parametres?);
liste_parametres : (parametre_fonction)*;
parametre_fonction : ':'! ID {tmp.addNomParam($ID.getText());
empileNomParam("PARAM_"+$ID.getText());};

exprOuAppel : expr | appelProc2;

appelProc2 : (ID^ valParams) { appelProc($ID.getText()); };
valParams : ((expr {empileValParam(0.0);})*)
          -> ^(VALUES (expr)*);
```

Tree-walker

```
tree grammar LogoTree;
options {
  tokenVocab = Logo;
  ASTLabelType=CommonTree;
}
@header {
  package logoparsing;
  import logogui.Traceur;
  import logogui.Log;
```

```
  import java.util.Map;
  import java.util.HashMap;
  import logogui.LogoContext;
  import logogui.Procedure;
  import java.util.Collection;
  import java.util.Iterator;
  import java.util.Vector;
}
@members{
  Traceur traceur;
  LogoContext context;

  private Vector<String> paramsNoms;
  private Vector<Double> paramsValues;
  private Procedure tmp;
  private int marker_fin; // fin bloc

  public void initialize(java.awt.Graphics g) {
    traceur = Traceur.getInstance();
    traceur.setGraphics(g);
  }
  public void mark() {
     ((CommonTreeNodeStream)input).mark();
  }
  public void push(int index) {
     ((CommonTreeNodeStream)input).push(index);
  }
  public void pop() {
     ((CommonTreeNodeStream)input).pop();
  }
  public void seek(int index) {
     ((CommonTreeNodeStream)input).seek(index);
  }
  public void rewind(int index) {
     ((CommonTreeNodeStream)input).rewind(index);
  }
  public int index() {
     return ((CommonTreeNodeStream)input).index();
  }

  // récupère la valeur de la variable name
  public double getVar(String name) {
    Double value = null;
      Stack pile = context.getPile();
      HashMap<String,Double> h;

      for(int i=pile.size()-1; i>=0; i--){
             h = (HashMap<String,Double>)pile.get(i);
             value = h.get(name);
             if ( value!=null ) {
                    if(i==pile.size()-1 && value.doubleValue() == -1.0){
                           Log.appendnl("variable locale "+name+" non initialisée");
                           return 0.0; // valeur par défaut
                    }
                           return value.doubleValue();
             }
      }
      return 0.0;
    }

    // ajoute une variable name avec pour valeur value au contexte courant
    public void setVar(String name, double value) {
      Double p = new Double(value);
       HashMap<String,Double> h = (HashMap<String,Double>)context.peek();
             h.put(name, p);
    }
```

```
    // ajoute une variable de contexte avec pour nom réservé "RENDS" et la valeur
de retour de la fonction
    // la variable est créée dans le contexte de l'appel pour etre utilisée ensuite
    public void setReturnValue(double value) {
       Double p = new Double(value);
       HashMap<String,Double> h =
(HashMap<String,Double>)context.getPile().get(context.getPile().size()-2); //
hashmap des variables du contexte appelant
             h.put("RENDS", p); // nom fixé pour la variable de retour
    }

    // récupère la valeur de retour de la fonction appelée
    public double getReturnValue() {
       Double p;
       HashMap<String,Double> h =
(HashMap<String,Double>)context.getPile().get(context.getPile().size()-1); //
hashmap des variables du contexte appelant
             p = h.get("RENDS"); // nom fixé pour la variable de retour
             if(p==null) return 0.0;
             else return p.doubleValue();
    }

    // ajoute un paramètre d'appel qui a pour valeur value
    public void empileValParam(double value){
       Double p = new Double(value);
       paramsValues.add(p);
    }

    // ajoute un nom de paramètre lors de la déclaration d'une procédure
    public void empileNomParam(String value){
       paramsNoms.add(value);
    }

}

prog [LogoContext scope]
@init{
  context = scope;
  context.init();
  paramsNoms = new Vector<String>();
}
@after{
  traceur.dessinerTortue();
}
:
  ^(PROGRAMME (instruction)* { marker_fin = input.mark(); })
{Log.appendnl("Programme principal");}
;

expr returns [double r=0] :
  ^(PLUS a=expr b=expr) {r = a+b;}
  | ^(MOINS a=expr b=expr) {r = a-b;}
  | ^(MULT a=expr b=expr) {r = a*b;}
  | ^(DIV a=expr b=expr) {r = a/b;}
  | REAL {r = new Double($REAL.getText()).doubleValue();}
  | ID { if($ID.getText() != "LOOP" && $ID.getText() != "RENDS") r =
getVar($ID.getText()); }
  | CAP { r = traceur.getCap(); }
  | ^(HASARD a=expr){r = traceur.hasard(a);}
  | LOOP { r = getVar("LOOP"); }
;

boolnexpr returns [boolean r=false] :
  ^(EGAL a=expr b=expr) { r = a == b; }
  | ^(LESSTHAN a=expr b=expr) { r = a < b; }
  | ^(MORETHAN a=expr b=expr) { r = a > b; }
  | ^(LESSOREQ a=expr b=expr) { r = a <= b; }
  | ^(MOREOREQ a=expr b=expr) { r = a >= b; }
```

```
;

instruction
@init{
  int mark_before_true = 0;
  int mark_before_false = 0;
  int marker_fin = 0;
  } :
   ^(AV a = expr) {traceur.avance(a);}
  | ^(TD a = expr) {traceur.td(a);}
  | ^(TG a = expr) {traceur.tg(a);}
  | ^(REC a = expr) {traceur.recule(a);}
  | ^(FCAP a = expr) {traceur.fixeCap(a);}
  | ^(FCC a = expr) {traceur.fixeCouleur(a);}
  | (VE) {traceur.videEcran();}
  | (MT) {traceur.montrerTortue();}
  | (CT) {traceur.cacherTortue();}
  | (LC) {traceur.leverCrayon();}
  | (BC) {traceur.baisserCrayon();}
  | ^(FPOS a = expr b = expr) {traceur.fixePosition(a,b);}
  | ^(DONNE ID a=expr){ if($ID.getText() != "LOOP" && $ID.getText() != "RENDS")
setVar($ID.getText(), a); }
  | ^(ECRIS g=exprOuAppel){ traceur.ecrire(g+""); }
  | ^(LOCALE ID){ if($ID.getText() != "LOOP" && $ID.getText() != "RENDS")
setVar($ID.getText(), -1.0); }
  | ^(IF e=boolnexpr {mark_before_true = input.mark();} . ({mark_before_false =
input.mark();} else_list=.)? )
       {
       HashMap<String,Double> h = new HashMap<String,Double>();
       if (e) {
              context.push(h);
              push(mark_before_true);
              list_instructions();
              pop();
              context.pop();
       }
       else if (else_list != null) {
              context.push(h);
              push(mark_before_false);
              list_instructions();
              pop();
              context.pop();
       }
       }
 | ^(REPETE a=expr {mark_before_true = input.mark();} .)
    {
    int cpt = (int)a;
    int nb_exec = 0;
    HashMap<String,Double> h = new HashMap<String,Double>();
       context.push(h);

       while(cpt > 0){
              setVar("LOOP", nb_exec); // pour récupérer le nombre de fois que le
REPETE => LOOP
              push(mark_before_true);
              list_instructions();
              pop();
              cpt--;
              nb_exec++;
       }
       context.pop();
    }

  | ^(WHILE {mark_before_true = input.mark();} condition=. {mark_before_false =
input.mark();} suite=.)
  {
       push(mark_before_true+1); // +1 sinon il prend le token <DOWN> arès le TANQUE
       boolean booleen = boolnexpr();
```

```
        pop();
        while (booleen) {
                // Répétition des instructions de la boucle
                push(mark_before_false);
                list_instructions();
                pop();
                // Test de condition d'exécution de la boucle suivante
                push(mark_before_true+1); // +1 sinon il prend le token <DOWN> arès le
TANQUE
                booleen = boolnexpr();
                pop();
        }
 }

 | ^(POUR ID {mark_before_true = input.mark();} . ({mark_before_false =
input.mark();} code=.)? {marker_fin = input.mark();} )
 {
        Procedure p = new Procedure($ID.getText());
        p.setFinBloc(marker_fin-1);

        if(code==null){ // pas de parametre
                p.setDebut(mark_before_true);
                tmp=p;
        }else{ // parametres
                p.setDebut(mark_before_false);
                tmp=p;

                // récupère les noms des paramètres
                push(mark_before_true);
                list_nomparams();
                pop();
        }
        context.addProcedure(p);
 }
 | appelProcedure
 | ^(RETURN a=expr)
 {
        setReturnValue(a);
        rewind(tmp.getFinBloc()); // stoppe la procédure
 }
;

appelProcedure returns [double valeurRetour=0]
@init{
  int mark_before_true = 0;
}
:
^(ID {mark_before_true = input.mark();} .)
 {
        paramsValues = new Vector<Double>();
        push(mark_before_true+1);
        list_valparams2();
        pop();

        Procedure p = context.getProcedure($ID.getText());
        tmp = p;
        int pos = p.getDebut();
        push(pos);
        // correspondance NOMS/VALEURS des paramètres
        // a mettre dans le contexte des variables
        HashMap<String,Double> h = new HashMap<String,Double>();

        Vector<String> noms = p.getNomsParams();
        for(int i=0; i<noms.size(); i++){
                String nom = noms.get(i);
                Double val = paramsValues.get(i);
                h.put(nom,val);
        }
```

```
	context.push(h);
	list_instructions_procedure(); // éxécution de la procédure
	context.pop();
	pop();

	// récupère la valeur de retour de la precédure appelée si elle existe / 0 
sinon
	valeurRetour = getReturnValue();
}
;

list_valparams2: ^(VALUES list_valparams);
list_valparams: (a=expr {
	empileValParam(a);}
)*;

list_nomparams: ^(PARAMS liste_noms_parametres);
liste_noms_parametres: (ID {
	tmp.addNomParam($ID.getText());}
)+;

list_instructions: ^('[' instruction+);
list_instructions_procedure: ^(INSTR instruction+);

exprOuAppel returns [double q=0] :
	a=expr {q=a;} |
	w=appelProcedure {q=w;}
;
```

Traceur

```
package logogui;
import java.awt.Color;
import java.awt.Graphics2D;
import java.awt.Graphics;
import java.awt.Image;
import java.awt.Toolkit;
import java.awt.geom.AffineTransform;
import java.awt.image.AffineTransformOp;
import java.awt.image.BufferedImage;
import java.io.File;
import java.io.IOException;
import java.util.HashMap;
import javax.imageio.ImageIO;

public class Traceur {
	private static Traceur instance;
	private Graphics2D g2d;
	private double initx = 300, inity = 300;   // position initiale
	private double posx = initx, posy = inity; // position courante
	private int angle = 90;
	private double teta;
	private boolean crayon = true;
	private boolean tortue = true;
	private int couleur = 0;
	private int cap = 0 ;
	private BufferedImage img; // fichier image de la tortue
	private int img_width;
	private int img_height;

	// tableau des couleurs possibles du crayon
	private Color tab_couleurs[] = 
	{Color.black, Color.red, Color.green, Color.yellow,  Color.blue, 
	 Color.magenta, Color.cyan, Color.white};

	public Traceur() {
		setTeta();
```

```
	}
	public static Traceur getInstance() {
		if (instance == null)
			instance = new Traceur();
		return instance;
	}
	public void setGraphics(Graphics g) {
		g2d = (Graphics2D) g;
	}
	private int toInt(double a) {
		return (int) Math.round(a);
	}
	public void avance(double r) {
		double a = posx + r * Math.cos(teta) ;
		double b = posy - r * Math.sin(teta) ;
		int x1 = toInt(posx);
		int y1 = toInt(posy);
		int x2 = toInt(a);
		int y2 = toInt(b);
		if(crayon) g2d.drawLine(x1,y1,x2,y2);
		posx = a;
		posy = b;
	}
	public void td(double r) {
		angle = (angle - toInt(r)) % 360;
		setTeta();
		cap = (cap + toInt(r)) % 360;
	}
	public void tg(double r) {
		angle = (angle + toInt(r)) % 360;
		setTeta();
		cap = (cap - toInt(r)) % 360;
	}
	private void setTeta() {
		teta = Math.toRadians(angle);
	}
	public void recule(double r) {
		avance(-r);
	}
	public void fixeCap(double r) {
		// r = 0 => la tortue se tourne pour aller vers le haut
		cap = toInt(r) % 360;

		if(r>=0 && r<=90){
			angle = (90 - cap) % 360;
		}else if(r>90 && r<=360){ // else
			angle = (450 - cap) % 360;
		}
		setTeta();
	}
	public void fixeCouleur(double r) {
		couleur = (toInt(r) % 8);
		g2d.setColor(tab_couleurs[couleur]);
	}
	public void videEcran() {
		g2d.clearRect(0, 0, 1000, 1000);
	}
	public void montrerTortue() {
		//TODO
		tortue = true;
	}
	public void cacherTortue() {
		//TODO
		tortue = false;
	}
	public void leverCrayon() {
		crayon = false;
	}
```

```
	public void baisserCrayon() {
		crayon = true;
	}
	public void fixePosition(double a, double b) {
		posx = a;
		posy = b;
	}
	public void ecrire(String s){
		g2d.drawString(s, (float)posx, (float)posy);
	}
	// renvoie le cap de la tortue
	public int getCap(){
		 return cap;
	}
	// renvoie un nombre au hasard entre 0 et n
	public double hasard(double n){
		int nb = new Float(Math.random()*n).intValue();
		return nb;
	}

	public void dessinerTortue(){
		// image de la tortue
		try {
			img = ImageIO.read(new File("src\\logogui\\tortue.png"));
		} catch (IOException e) {
			e.printStackTrace();
		}
		img_width = img.getWidth(null);
		img_height = img.getHeight(null);

		// rotation en fonction du cap si besoin
		AffineTransform tx = new AffineTransform();
		tx.rotate(Math.toRadians(cap), img_width/2, img_height/2);
		AffineTransformOp op = new AffineTransformOp(tx,
AffineTransformOp.TYPE_BILINEAR);
		img = op.filter(img, null);
		if(tortue) g2d.drawImage(img, (int)(posx-img_width/2), (int)(posy-
img_height/2), null);
	}

}
```

Procedure

```
package logogui;
import java.util.Vector;

public class Procedure {
	private String nom;
	private int debut = 0;
	private Vector<String> nomsParams;
	private int fin_bloc = 0;

	public Procedure(){
		nomsParams = new Vector<String> ();
		nom = "";
	}
	public Procedure(String name){
		nomsParams = new Vector<String> ();
		nom = name;
	}
	public void setDebut(int d){
		debut = d;
	}
	public int getDebut(){
		return debut;
	}
```

```
	public String getName(){
		return nom;
	}
	public void setFinBloc(int i){
		fin_bloc = i;
	}
	public int getFinBloc(){
		return fin_bloc;
	}
	public void addNomParam(String param){
		int old = nomsParams.indexOf(param);
		if(old == -1) nomsParams.add(param);
		else nomsParams.setElementAt(param, old); // remplace ancienne
valeur si elle existait déja
	}
	public Vector<String> getNomsParams(){
		return nomsParams;
	}
	public int getNbParams(){
		return nomsParams.size();
	}
	public void afficher(){
		System.out.println("---> Procedure "+nom+" : debut="+debut+" -
nomsParams="+nomsParams);
	}
}
```

LogoContext

```
package logogui;
import java.util.HashMap;
import java.util.Stack;
import java.util.Vector;

public class LogoContext {
	private Stack <HashMap<String,Double>> pile; // variables
	private Vector<Procedure> listeProcedures; // procédures

	public void init() {
		pile = new Stack <HashMap<String,Double>> ();
		listeProcedures = new Vector<Procedure> ();

		// contexte global au programme
		HashMap<String,Double> h = new HashMap<String,Double> ();
		pile.push(h);

	}
	public Stack<HashMap<String,Double>> getPile(){
		return pile;
	}
	public Vector<Procedure> getListeProcedures(){
		return listeProcedures;
	}
	public void push(HashMap<String,Double> h){
		pile.push(h);
	}
	public void pop(){
		pile.pop();
	}
	public HashMap<String,Double> peek(){
		return pile.peek();
	}
	public Procedure getProcedure(String name){
		Procedure p;
		for(int j=0; j<listeProcedures.size(); j++){
```

```
                    p = (Procedure)listeProcedures.get(j);
                    if(p.getName().compareTo(name) == 0){ // fonction trouvée
                          return p;
                    }
             }
             return null; // pas trouvé
      }
      // cherche la variable name dans la liste
      // des noms de paramètres des procédures
      public boolean getVarInProcedures(String name){
             Procedure p;
             for(int j=0; j<listeProcedures.size(); j++){
                    p = (Procedure)listeProcedures.get(j);
                    if(p.getNomsParams().contains(name)){
                          return true; // trouvé
                    }
             }
             return false; // pas trouvé
      }
      public void addProcedure(Procedure p){
             int old = listeProcedures.indexOf(p);
             if(old == -1) listeProcedures.add(p);
             // remplace ancienne valeur si elle existait déja
             else listeProcedures.setElementAt(p, old);
      }
}
```