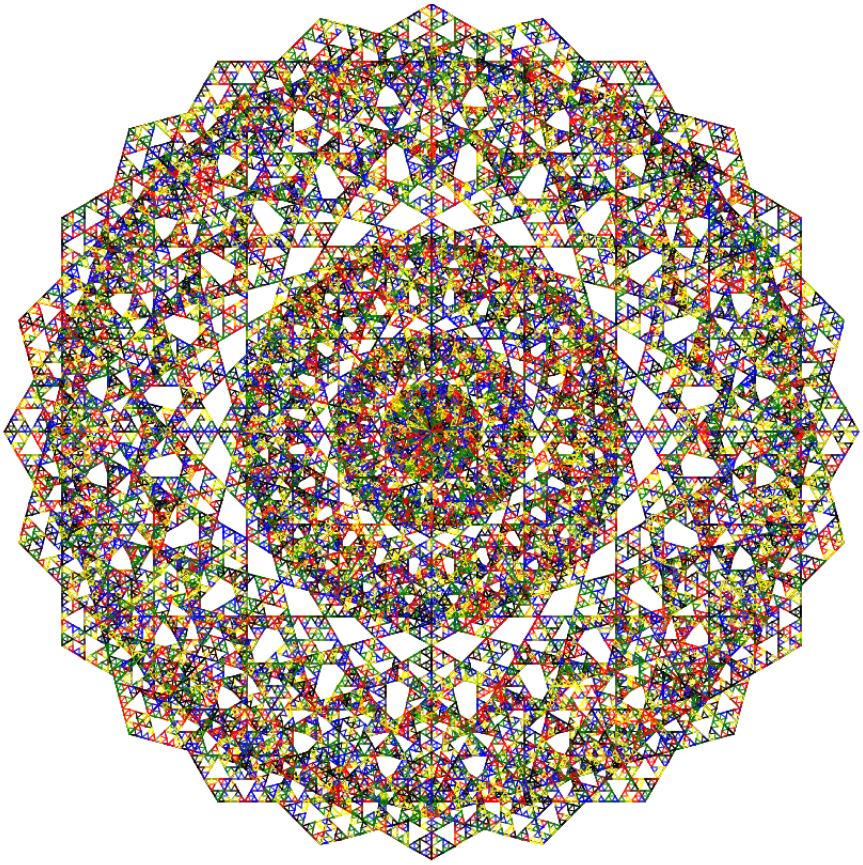
Université de Technologie de Compiègne

NF11 – Théorie des langages de programmation

Projet P17

Implémentation d'un langage type Logo



Trouve Robin – Duchemin Quentin

06 Juin 2017

RÉSUMÉ

L'ensemble du projet est disponible sur GitHub. Note : pour un affichage cohérent, il faut mettre le programme en **plein écran**. Ceci permet de visualiser entièrement la figure d'exemple.

Le projet NF11 P17 vise à créer un dérivé du langage Logo permettant en particulier d'utiliser des instructions graphiques simples pour dessiner des formes.

Le langage ainsi créé permet d'effectuer plusieurs opérations graphiques (avancer, reculer, changer de couleur...), d'évaluer des expressions arithmétiques, d'utiliser des conditions, des boucles, du hasard, ainsi que des procédures ou des fonctions (possiblement récursives).

Le projet s'appuie sur la bibliothèque Java ANTLR, un générateur d'analyseur syntaxique descendant, *i.e.* LL(*) généralement utilisé pour construire des langages de programmation.

Le langage créé est décrit par une grammaire au format `g4`, spécifique à ANTLR. Elle n'a pas nécessairement besoin d'être LL(*) depuis sa version 3, ANTLR gérant par exemple la récursivité à gauche.

À partir de cette grammaire sont générés un ensemble de classes Java permettant :

- D'une part, de construire l'arbre de dérivation du programme à analyser ;
- D'autre part, d'utiliser le pattern Visitor pour parcourir l'arbre et prendre des décisions.

Dans cette version du langage (contrairement à Logo), chaque fonction possède sa propre pile, n'autorisant pas les variables globales mais préservant l'état des variables locales lors des appels récursifs.

TABLE DES MATIÈRES

1 GRAMMAIRE 1

2 STRUCTURES DE DONNÉES 3
2.1 Structures de base 3
2.2 Gestion des variables 3
2.3 Gestion des boucles 4
2.4 Gestion des fonctions et des procédures 5

3 FLUX D'EXÉCUTION D'UN APPEL DE FONCTION 7

4 EXEMPLE 9

1 GRAMMAIRE

La figure 1 montre la grammaire décrivant notre langage Logo.

Un programme Logo est constitué d'un ensemble optionnel de déclarations de fonctions et d'une liste d'instructions. Une déclaration comporte un nom et une liste de paramètre. Son corps est aussi une liste d'instructions, et la présence du mot clé `rends` conditionne le fait que ce soit une *fonction* ou une *procédure*.

Une liste d'instructions est composée d'au moins une instruction. Une instruction peut être de quatre types :

- Instruction **graphique** : elle permet d'agir sur le dessin (avancer, reculer, changer la couleur…) ;
- Instruction de **contrôle** : elle permet d'agir sur le fil d'exécution du programme, par des branchements (conditions) ou des répétitions (tant que) ;
- Instruction d'**appel** : elle permet d'appeler une procédure ;
- Instruction d'**affectation** : elle permet d'associer un symbole à une valeur numérique.

La plupart de ces instructions ont un point commun : elles agissent sur des **expressions**. Les expressions sont un ensemble de symboles qui appellent à une **évaluation**. Une fois évalués, ces symboles produisent une **littérale numérique**. Les expressions sont de trois types :

- Expression **arithmétique** : il s'agit de calculs, ou de génération de nombres pseudo-aléatoires ;
- Expression d'**appel** : elle permet d'appeler une fonction et s'évalue en l'*expression de retour* de la fonction ;
- Expression de **résolution** : elle comporte un symbole et s'évalue comme la valeur de ce symbole, s'il est affecté.

Le langage ainsi décrit permet d'écrire des programmes assez complets, incluant en particulier les procédures récursives.

```
grammar Logo;

@header {
   package logoparsing;
}

INT : '0' | [1-9][0-9]*;
NOM : [a-z]+;
WS : [ \t\r\n]+ -> skip ;


programme :
    (declaration)*
    liste_instructions
;
liste_instructions :
    (instruction)+
;

declaration :
    'pour' NOM (':'NOM)* liste_instructions ('rends' exp)? 'fin'
;

instruction :
    'av' exp                                  # av
  | 'td' exp                                  # td
  | 'tg' exp                                  # tg
  | 'lc'                                      # leve
  | 'bc'                                      # pose
  | 've'                                      # clear
  | 're' exp                                  # recule
  | 'fcc' exp                                 # couleur
  | 'fpos' exp exp                            # position
  | 'repete' exp '[' liste_instructions ']'   # repete
  | 'donne' '"'NOM exp                        # donne
  | 'si' expbool '[' liste_instructions ']' ('[' liste_instructions ']')? # si
  | 'tantque' expbool '[' liste_instructions ']' #tantque
  | NOM exp*                                  # appelprocedure
;

exp:
    ':'NOM                                    # var
  | 'hasard' exp                              # hasard
  | exp ('*' | '/') exp                       # mul
  | exp ('+' | '-') exp                       # sum
  | INT                                       # int
  | 'loop'                                    # loop
  | '(' exp ')'                               # parenthese
  | NOM '(' exp* ')'                          # appelfonction
;

expbool:
    exp ('<' | '>' | '>=' |'<=' | '=' | '!=') exp   # cond
  | expbool 'ET' expbool                              # et
  | expbool 'OU' expbool                              # ou
;
```

FIGURE 1 – Grammaire de notre langage Logo

2 STRUCTURES DE DONNÉES

2.1 STRUCTURES DE BASE

Pour ce projet, il a été nécessaire d'implémenter différentes structures ou type de données, afin de répondre aux demandes et d'implémenter une logique qui nous paraisse cohérente.

Tout d'abord, la grammaire définie au préalable permet de générer un arbre à partir du programme d'entrée, grâce au *parser* généré par ANTLR. Cet arbre syntaxique forme la base de l'analyse du programme. Il faut alors visiter l'arbre – en étendant la logique de base implémentant le pattern Visitor fournie par ANTLR. Chaque nœud devant produire une **évaluation** doit avoir les moyens de la faire remonter à son parent.

À cette fin, une structure de données de base nous est fournie par ANTLR : il s'agit de la propriété du type `ParseTreeProperty` qui va nous permettre d'attribuer une valeur d'un certain type (dans notre implémentation, `Double`) à chaque nœud de l'arbre. Chaque visiteur va pouvoir attribuer une valeur pour le nœud qui lui correspond, tandis que des nœuds parent va pouvoir consulter les valeurs de leurs nœuds fils. Il s'agit essentiellement d'une sorte de tableau associatif avec pour clé un élément de type `ParseTree` (ou enfant) et pour valeur un élément de type `Double`.

A cette structure centrale, nous avons rajouté deux piles, une `Map` ainsi que deux classes afin d'étendre les possibilités de notre langage : c'est ce qui fait l'objet des prochains paragraphes.

2.2 GESTION DES VARIABLES

Les variables répondent à des besoins spécifiques. D'abord, on doit pouvoir les initialiser, leur affecter une valeur et la récupérer. Ensuite, il est préférable de gérer la **portée** (ou *scope*) des variables afin, d'une part, d'éviter les variables globales, et d'autre part, d'interdire l'accès aux variables locales d'une procédure depuis l'extérieur.

Pour pouvoir gérer ces multiples besoins concernant les variables, l'implémentation peut être séparée en deux parties : d'abord avec une classe `SymbolTable`, qui va représenter la table des symboles, c'est-à-dire les variables et leurs valeurs. Elles sont stockées dans une `Map` – le nom de la variable étant la clé.

Ensuite, avec l'arrivée des blocs de codes séparés que sont les procédures et les fonctions, il a été nécessaire de modifier l'implémentation pour ne pas avoir des variables globales, mais bien des variables locales à chaque fonction. L'implémentation doit autoriser l'existence simultanée de deux variables liées au même symbole à des portées différente.

Pour ce faire, on garde en propriété de la classe qui visite l'arbre une **pile de table de symboles**, sous la forme `Stack<SymbolTable>`. Son fonctionnement est simple : une première table des symboles est ajoutée à l'initialisation du programme. Elle contient les variables du programme principal. À chaque appel de fonction ou de procédure, on empile une table de symboles supplémentaire, qui va contenir les variables locales au bloc ; similairement, on la dépile lorsque la procédure ou la fonction termine. Pour accéder aux variables il suffit alors de consulter la table des symboles au sommet de la pile, qui contient les variables correspondant au bloc courant.

Ce système peut trouver une analogie dans la gestion mémoire réelle des variables, dans un programme C par exemple. Même au niveau assembleur, chaque « table des symboles » est délimitée par un *cadre de pile* et un *pointeur de pile*, qui définissent l'espace mémoire réservé aux variables locales.

2.3 GESTION DES BOUCLES

Nous avons précédemment parlé de **blocs** pour définir les fonctions et les procédures, mais elle n'en forment qu'une sous-catégorie. Le deuxième type de bloc implémenté par le langage est la structure de **boucles**, qui possède au moins une variable locale : le **compteur de boucle**.

La première structure choisie pour les boucles était l'utilisation d'un simple entier stockant la valeur de la boucle courante. Cependant, cette idée pose un problème en tant qu'elle n'autorise pas l'imbrication de boucles à compteurs indépendants.

La structure adoptée finalement est une **pile de compteurs** (sous la forme `Stack<Integer>`), afin de pouvoir suivre l'exécution des portions de boucles. À chaque nouvelle itéra-

tion, le compteur sur le sommet de la pile est mis à jour, puis est dépilé à la fin de la dernière itération.

2.4 GESTION DES FONCTIONS ET DES PROCÉDURES

Les procédures et les fonctions doivent être stockées après analyse de leur déclaration, car on doit pouvoir y faire référence ultérieurement.

À ce titre, nous avons créé la classe `Procedure`, qui, contrairement à son nom, peut décrire à la fois des procédures et des fonctions. Il s'agit d'une classe qui contient les informations nécessaires pour un appel : un booléen pour savoir si c'est une fonction ou une procédure et une liste des noms de paramètres que la fonctions prend en entrée. Cette liste sera utilisée à chaque appel de la fonction pour remplir la table des symboles avec les arguments effectifs.

De plus, chaque objet `Procedure` comprend les instructions à exécuter, et enfin, pour les fonctions, l'expression à retourner.

Pour pouvoir accéder aux fonctions et procédures déclarées en début de programme, on les ajoute un tableau associatif (sous la forme `Map<String, Procedure>`), la clé étant le nom de la procédure et la valeur une instance de la classe décrite précédemment.

3 FLUX D'EXÉCUTION D'UN APPEL DE FONCTION

Nous avons déjà expliqué brièvement comment se passe l'analyse du programme et son exécution : chaque nœud de l'arbre peut produire une valeur utilisée par ses parents, et le pattern Visitor permet de sélectionner le code à exécuter selon le type de nœud. Ainsi, une expression de type `av 5 * 3` appellera la méthode de visite correspondant à l'avancement ; elle-même appellera la méthode de visite de ses nœuds fils (ici, la méthode de multiplication, qui évaluera les entiers, puis stockera le résultat de l'opération).

Le cas d'un appel de fonction, par exemple, demande plus de traitement. Tout d'abord, une fonction prend en général un ou plusieurs arguments. Le traitement des arguments est similaire à celui d'un nœud standard : on visite les expressions filles et les évalue en littérales numériques.

Le traitement change ici : il faut donner le moyen à la fonction qui sera appelée de disposer de ses paramètres. On créé ainsi une table de symboles propre à cette fonction, et on associe dans l'ordre le nom de chaque paramètre à la valeur évaluée par la visite. Le comportement est similaire à des affectations successives. En cas d'incompatibilité sur le nombre de paramètres, l'appel à la fonction est ignoré et un message d'erreur est affiché.

La table de symboles ainsi créée est **empilée**, et le nœud correspondant à la liste d'instructions de la fonction est visité. Il en va ainsi de suite si d'autres appels (récursifs ou non) sont présents.

Enfin, une fois l'ensemble des instructions visitées et exécutées, l'instruction de **retour** est visitée et son évaluation est sauvegardée comme la valeur du nœud courant, pour utilisation postérieure par les appelants. La table de symboles est finalement dépilée.

Notons qu'afin de mettre en place la récursion correctement, il faut faire attention à un détail. Si un appel récursif comporte une opération binaire, il faut récupérer la valeur de chaque nœud dans le `ParseTreeVisitor` **immédiatement** après son évalua-

tion. Si on le fait **après** les deux évaluations, toutes les valeurs produites par les appels récursifs seront écrasés. En effet, `ParseTreeVisitor` fonctionnant grâce au `hashCode` pour comparer les clés, et les arbres à visiter étant les mêmes au sens Java au fil de la récursion, les valeurs ne sont pas conservées.

4 EXEMPLE

La figure 2 montre le résultat d'un programme écrit dans notre dérivé de Logo. Il s'agit de la superposition de 24 triangle de Sierpiński de degré 400 (*i.e.* avec 400 itérations de l'algorithme), chaque triangle étant dessiné avec une rotation de 15 degrés (pour décrire un cercle).

La procédure `triangle` permet de construire un unique triangle de Sierpiński, chaque petit triangle ayant une couleur aléatoire.

Le programme principal se contente d'appeler plusieurs fois cette procédure en orientant le sens du dessin après chaque appel.

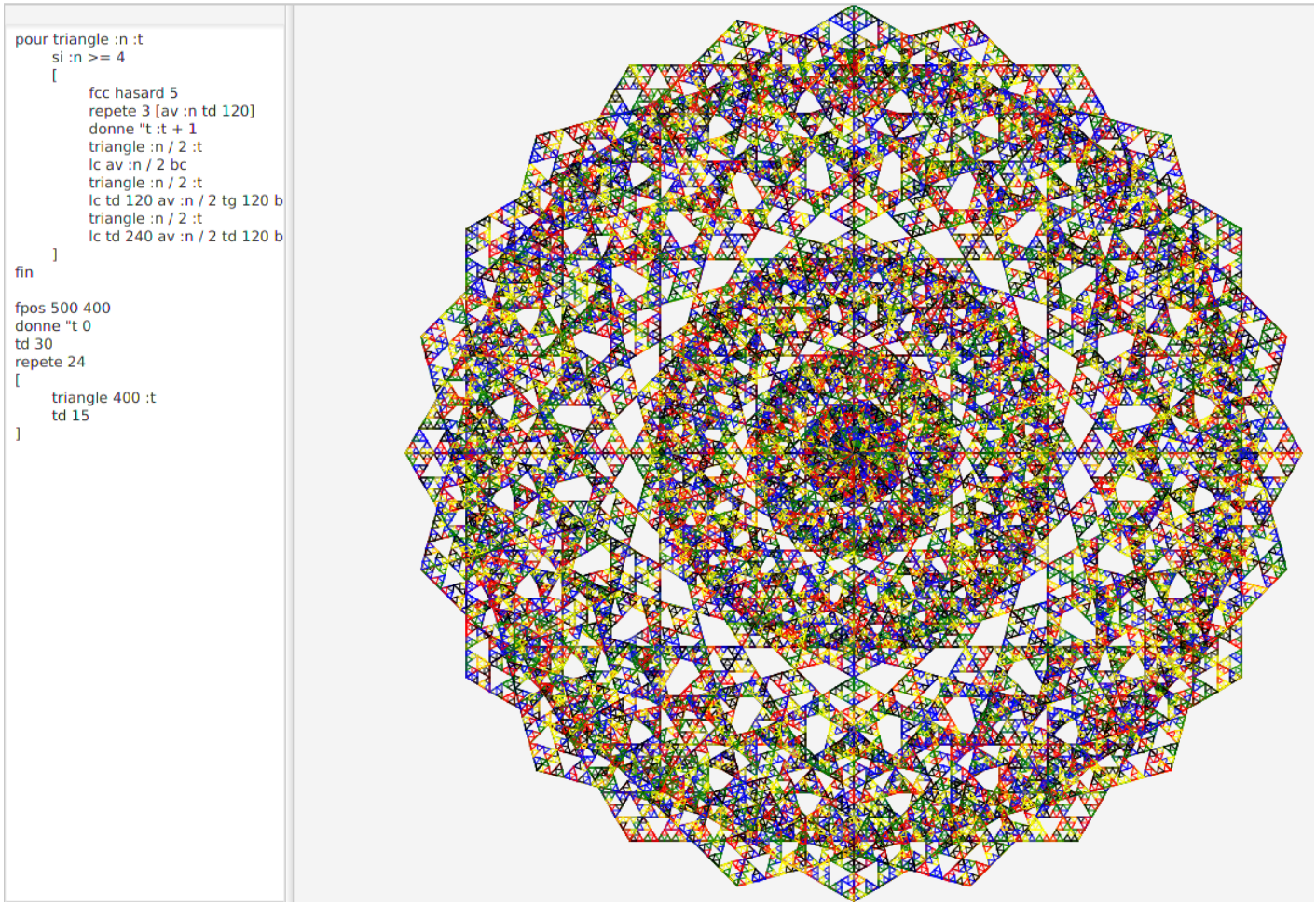


FIGURE 2 – Exemple de dessin obtenu à partir de notre langage Logo.