# Rapport NF11
## Projet LOGO

Impellettieri Florian
Hammi Marouane

14 juin 2016

# Table des matières

# 1 Introduction

## 1.1 Contexte

Ce document constitue le rapport du projet de NF11. Au cours de ce projet l'objectif a été de réaliser une grammaire dans le langage LOGO.

# 2 Grammaire

Ci-dessous figure la grammaire de notre projet, suivie d'une description synthétique des différents éléments la composant.

```
1  grammar Logo;
2
3  @header {
4    package logoparsing;
5  }
6
7  INT : '0' | [1-9][0-9]* ;
8  ID : [A-Za-z][A-Za-z0-9]* ;
9  SIGN : ('-'|'+') ;
10 WS : [ \t\r\n]+ -> skip ;
11 exp : exp ('*'|'/') exp #mult
12     | exp ('+'|'-') exp #sum
13     | ID '('(exp)*')'   #appelFonction
14     | atom              #arule
15     ;
16
17 atom : INT              #int
18      | '(' exp ')'      #parent
19      | 'hasard' exp     #hasard
20      | 'loop'           #loop
21      | ('-'|'+') INT    #sigInt
22      | ':'ID            #variable
23      ;
24
25 expbool : '!'expbool                          #logiqueNegation
26         | expbool '&' expbool                  #logiqueEt
27         | expbool '|' expbool                  #logiqueOu
28         | '(' expbool ')'                      #logiqueParent
29         | exp ('<' | '>' | '<='| '>=' | '!=' | '==') exp #boolOperation
30         ;
31
32
33 programme : methodes? liste_instructions
34 ;
35
36 methodes :
37   (pour)+
38 ;
39
40 pour :
41   'pour' ID (':'ID)* (liste_instructions)? (rends)? 'fin'
42 ;
```

```
43 rends :
44  'rends' exp
45 ;
46
47 liste_instructions :
48   (instruction)+
49 ;
50
51 instruction :
52     'av' exp # av
53   | 'td' exp # td
54   | 'tg' exp # tg
55   | 're' exp # re
56   | 'fpos' '[' atom  atom ']' # fpos
57   | 'lc' # lc
58   | 'bc' # bc
59   | 've' # ve
60   | 'fcc' exp # fcc
61   | 'repete' exp '[' liste_instructions ']' #repete
62   | 'donne' '"'ID exp #donne
63   | 'si' expbool '[' liste_instructions ']' ('[' liste_instructions ']')? #si
64   | 'tantque' expbool '[' liste_instructions ']' #tantque
65   | ID '('(exp)*')' #appelPour
66
67 ;
```

Cette grammaire constitue la base du projet.

## 2.1 Descriptions expression

Les expressions sont constitué des différentes opérations et de mots clefs permettant de retourner une valeur (ex : hasard). Les appels de fonctions sont aussi contenus dans les expressions puisque ces derniers doivent retourner une valeur. Il existe aussi les expressions booléenne qui renvoient false ou true (0 ou 1). Que se soit les expressions booléennes ou les expressions arithmétique, l'ordre est étudié pour prendre en compte la priorité des opérateurs. Par exemple l'opérateur de multiplication est prioritaire par rapport à celui d'addition.

## 2.2 Descriptions instructions

Les instructions correspondent aux directives sont appelées dans le programme. Elles permettent d'effectuer des actions comme par exemple d'avancer de 100 (av 100).

Il existe différents types d'instructions :

- celles permettant d'agir sur le traceur (av, re, lc, fpos,...)
- les boucles : 'repete' et 'tantque'
- l'instruction 'donne' qui permet d'affecter à une variable une valeur
- l'instruction de condition 'si', qui donne la possibilité de tester une expression booléenne et d'agir en conséquence
- l'instruction qui permet de réaliser l'appel d'une procédure, qui a été défini

### 2.3 Descriptions méthodes

Il est possible de définir une ou plusieurs méthodes que se soit des fonctions ou des procédures. La définition des méthodes doit être faite avant le programme principal.

Les fonctions doivent renvoyer une valeur, pour ce faire elles doivent utiliser le mot-clef 'rends' qui correspond au terme return dans la plupart des langages, excepté qu'il doit être utilisé juste avant le mot-clef 'fin'. Puisqu'une fonction doit renvoyer une valeur, on considère que l'appel de la fonction est une expression. L'appel d'une procédure est traitée comme une instruction normale.

Il est possible de passer aux méthodes plusieurs paramètres. Pour éviter toute ambiguïté il est nécessaire de passer les paramètres de la méthodes entre parenthèses.

## 3 Structure de données

Concernant les structures de données principales du projet, on peut référencer la Pile de Table des variables, la Pile Répète et la Table des Procédures.

### 3.1 Pile de Table des Variables

Une Table des variables est composée d'une map avec comme clef l'identifiant de la variable associé à la valeur Double de la variable. Ainsi lorsque l'identifiant de la variable est appelé dans le programme il est possible de retrouver sa valeur efficacement.

Initialement, il n'existait qu'une seule table des variables. Cependant lorsque les procédures et fonctions ont été mises en place il fallait trouver un moyen de gérer la portée des variables. Le choix a donc été réalisé de limiter la portée des variables dans le programme principal et dans les méthodes.

Ainsi, il existe une pile contenant initialement la Table du programme principal. Dès qu'une procédure ou une fonction est appelée une nouvelle Table est ajoutée dans la pile. Lorsque l'on recherche la valeur d'une variable donnée, il faut observer uniquement la Table se situant sur le dessus de la pile. Ce qui permet de s'assurer qu'une variable ne sera pas utilisée en dehors de sa portée définie.

Lorsque le traitement de la fonction ou de la procédure est finie, il faut retirer la Table de la pile pour revenir au contexte précédent.

### 3.2 Pile Répète

Cette pile permet de conserver le nombre d'itérations qui ont été effectué. Ce qui permet de récupérer cette valeur avec l'expression 'loop'. Quand bien même des instructions 'répète' seraient imbriquées.

À chaque itération de la boucle, l'index d'itération est ajouté en haut de la pile, puis les instructions

sont exécutés. Lorsque les instructions sont finis la valeur en haut de la pile est retirée, puis si la boucle n'est pas finie il faut recommencer.

### 3.3 Table des Procédures

Les procédures (et fonctions) posent un nouveau problème, puisque leurs définitions sont réalisées avant le programme principal, mais elles ne sont exécutées que lors de l'appel de ces derniers.

La solution adoptée, est de stocker dans une Map l'identifiant de la procédure comme clef. La valeur associé dans la Map correspond à la classe Procédure.

La classe Procédure possède 3 attributs :

- les listes des instructions, pour pouvoir visiter les nœuds fils de la procédure et les exécuter.
- une liste des identifiants des différents paramètres de la procédure.
- un nœud Rends, qui n'est utilisé que par les fonctions, afin de récupérer la valeur renvoyée par le mot clef 'rends', pour que la fonction retourne cette valeur.

Ainsi avec ces différentes informations, il est possible d'exécuter une procédure lorsqu'un appel de cette dernière est réalisée.

## 4 Principe de fonctionnement de la pile d'exécution

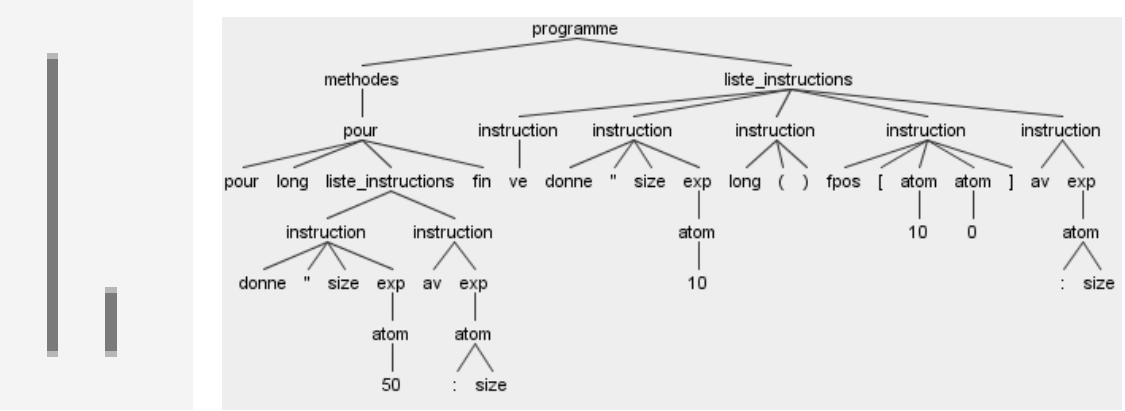
## 5 Principe de recherche d'une variable

```
1 pour long
2    donne size 50  av :size
3 fin
4 ve
5 donne size 10
6 long()
7 fpos [10 0] av :size
```

## 6 Principe expression 'loop'

L'expression 'loop' donne la possibilité de récupéré le nombre d'itérations qui ont été effectué dans la boucle 'répète'. Initialement, il y avait une variable au niveau du LogoTreeVisitor qui stockait la valeur de l'itération de la boucle 'répète'. Il ne restait plus qu'à retourner cette valeur lorsque 'loop' était appelé. Cette technique fonctionnait, mais présentait un inconvénient majeur dès que des boucles 'répète' étaient imbriquées. Puisqu'il n'existait qu'une seule variable pour les différentes boucles, une fois remonté dans la hiérarchie de l'arborescence des boucles, il n'était pas possible de retrouver la valeur de l'itération de la boucle parente.

Afin de résoudre ce problème, une pile a été créée. Dès qu'une boucle commence la valeur de l'itération




(a) Dessin (b) Arbre

FIGURE 1 – Exemple de recherche d'une variable

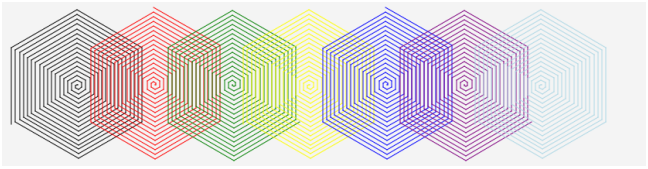
est ajouté au dessus de la pile, les instructions sont exécutées, et avant la prochaine itération la valeur au dessus de la pile est retirée. Ainsi, lorsque 'loop' observe la valeur au sommet de la pile, il s'agira de la valeur de l'itération de la boucle courante.

Les boucles peuvent être imbriquées sans perdre le compte dans le nombre des itérations.

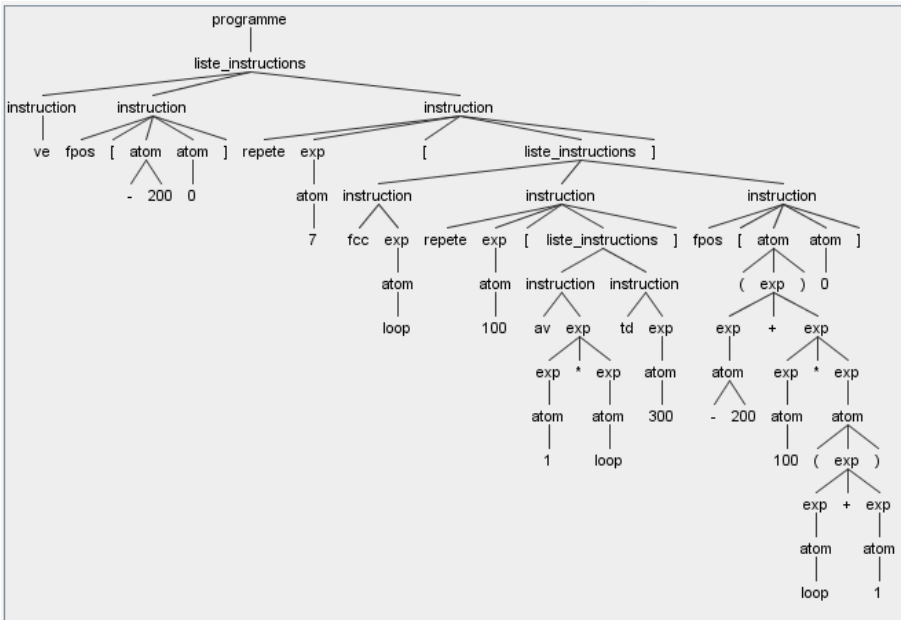
```
1  ve
2  fpos [-200 0]
3  repete 7
4  [
5     fcc loop
6     repete 100
7     [
8        av 1*loop
9        td 300
10    ]
11    fpos [(-200+100*(loop+1)) 0]
12 ]
```

# 7 Principe retour des fonctions

```
1  pour draw :cote :couleur
2     fcc :couleur
3     repete 6
4     [ av :cote     td 360/6 ]
5  fin
6  pour doubleDraw :n :iteration
7     donne t  :n*2
8     si :iteration > 0
9        [draw (
10          doubleDraw (   :t  (:iteration - 1))
11           :iteration )
12       ]
13    rends :t
```



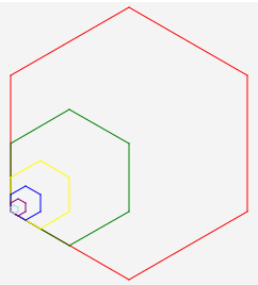
(a) Dessin



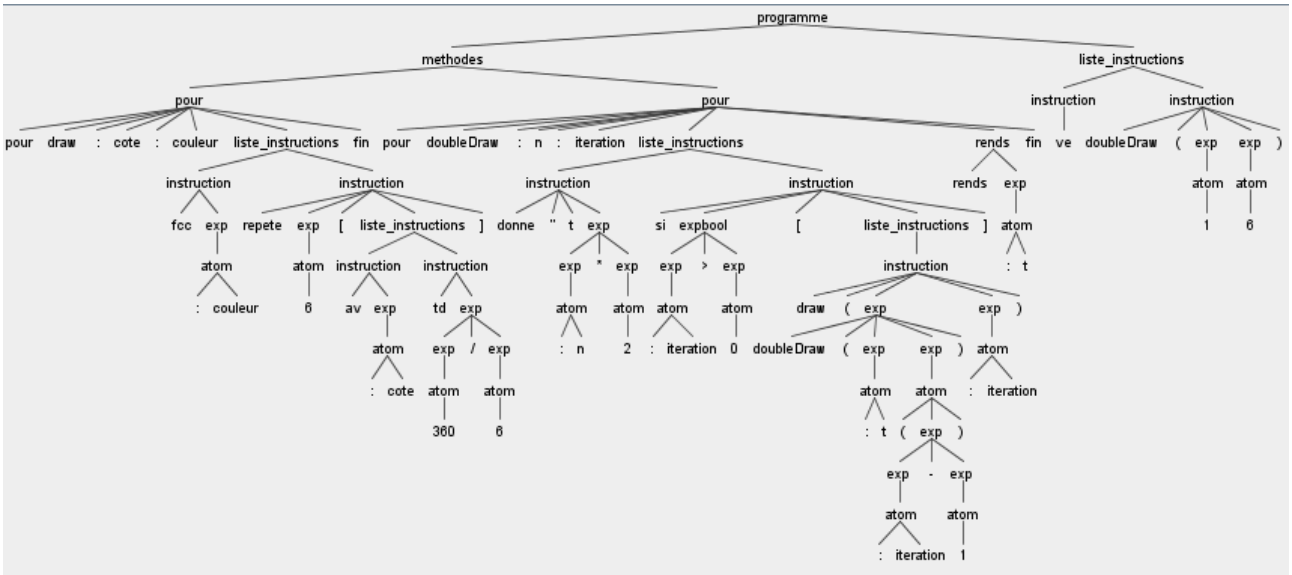

(b) Arbre

FIGURE 2 – Exemple repete imbriquées avec utilisation de loop

```
14 fin
15 ve
16 doubleDraw (1 6)
```



(a) Dessin




(b) Arbre

FIGURE 3 – Exemple fonctions récursives