

RAPPORT NF11

Projet LOGO

IMPELLETTIERI Florian
HAMMI Marouane

17 juin 2016

Table des matières

1	Introduction	3
1.1	Contexte	3
2	Grammaire	3
2.1	Descriptions expression	4
2.2	Descriptions instructions	4
2.3	Descriptions méthodes	5
3	Structure de données	5
3.1	Pile de Table des Variables	5
3.2	Pile Répète	5
3.3	Table des Procédures	6
4	Principe de fonctionnement de la pile d'exécution	6
5	Principe de recherche d'une variable	6
6	Principe expression 'loop'	7
7	Principe retour des fonctions	8

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

```

```

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

```

Cette grammaire constitue la base du projet.

2.1 Descriptions expression

Les expressions sont constituées des différentes opérations arithmétiques 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 qui 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 avant le programme principal

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 méthode 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 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 exécutées. Lorsque les instructions sont finies 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 les stocker dans une Map. En utilisant l'identifiant de la procédure comme clef et sa valeur associée qui correspond à une instance de 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.

Une procédure doit pouvoir recevoir des paramètres pour réaliser ses traitements. Lors de sa définition, on récupère les identifiants des paramètres que l'on place dans la liste des paramètres (List<String>). Puis lors de l'appel de la procédure, pour chaque paramètre on crée une variable dans la Table des Variables avec la valeur passée en arguments de la procédure.

Les paramètres sont considérés comme des simples variables une fois le traitement de la procédure commencé. Le nombre d'arguments est testé pour vérifier qu'il y ait le nombre de paramètres requis.

Pour éviter toute ambiguïté lors de l'appel d'une procédure ou d'une fonction, il faut passer entre parenthèses les arguments.

4 Principe de fonctionnement de la pile d'exécution

5 Principe de recherche d'une variable

Dans l'exemple ci-dessous on affecte à une variable size une valeur à deux reprises. Une fois dans le programme principale et une deuxième fois dans la procédure longueur().

En réalité la variable n'est définie qu'une seule fois pour chacune des portée. Puisque comme nous l'avons vu précédemment une Table des Variables est créée pour chaque nouvelle procédure (la dernière Table de la pile est celle du programme principal).

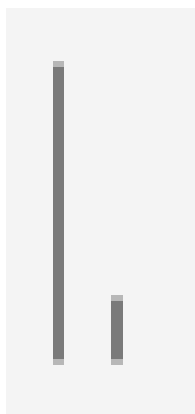
Pour affecter une valeur à une variable on utilise l'instruction 'donne "' suivie du nom de la variable et de sa valeur. La variable est inséré dans la Table du sommet de la pile.

Afin de retrouver la variable lorsque la variable est précédée de ' : ' dans le programme, il suffit de regarder dans la Table au sommet de la pile qui correspond à une Map. L'identifiant de la variable permet de récupérer la valeur pour pouvoir l'utiliser ailleurs.

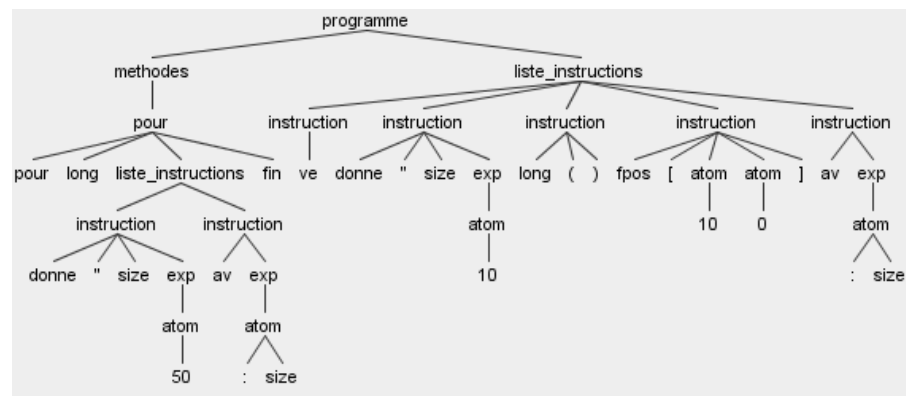
```

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

```



(a) Dessin



(b) Arbre

FIGURE 1 – Exemple de recherche d'une variable

6 Principe expression 'loop'

L'expression 'loop' donne la possibilité de récupérer le nombre d'itérations qui ont été effectués 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 est ajoutée au dessus de la pile, les instructions sont exécutées, et avant la prochaine itération la valeur au dessus de la pile est retirée.

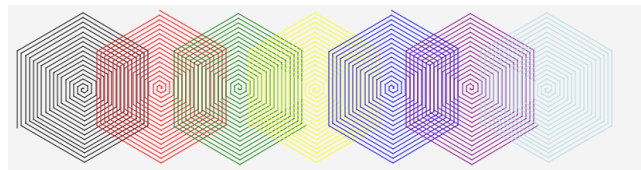
Ainsi, lorsque 'loop' observe la valeur au sommet de la pile, il s'agira de la valeur de l'itération de la boucle courante.

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

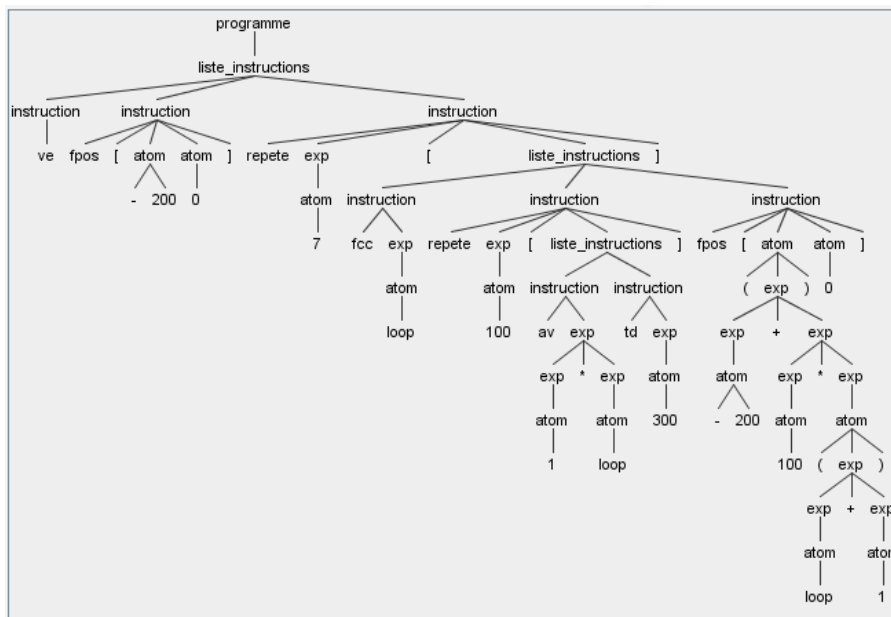
```

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 ]

```



(a) Dessin



(b) Arbre

FIGURE 2 – Exemple repete imbriquées avec utilisation de loop

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

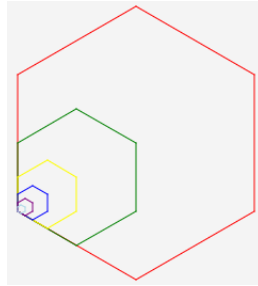
```



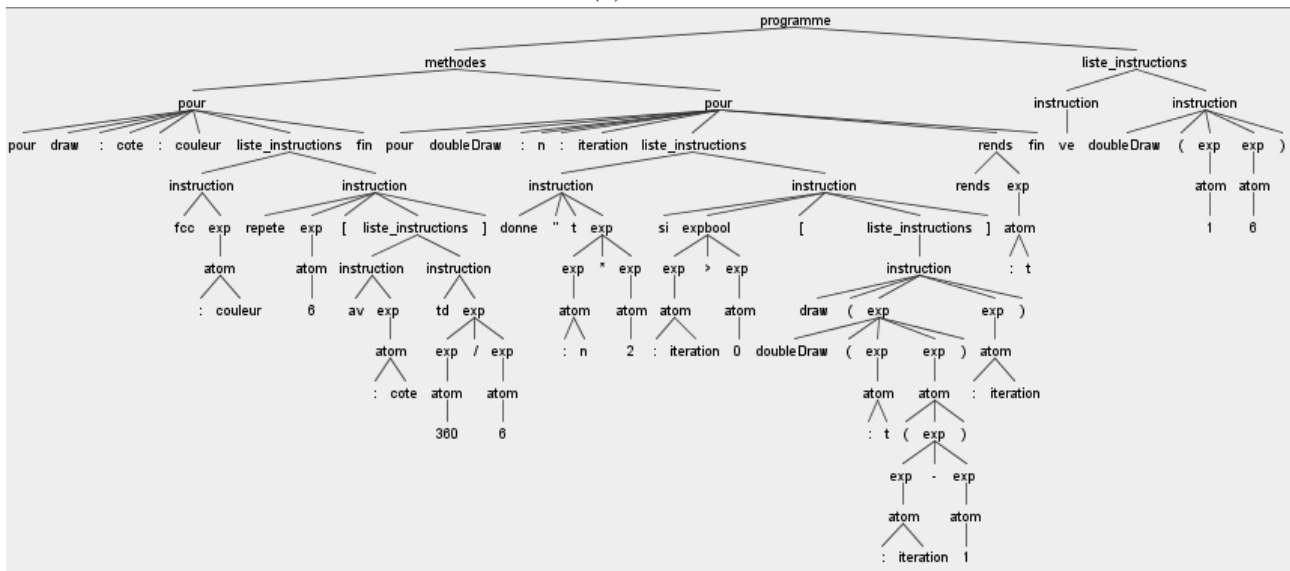
```

7  |  donne t :n*2
8  |  si :iteration > 0
9  |  [draw (
10 |      doubleDraw ( :t (:iteration - 1))
11 |      :iteration )
12 |  ]
13 |  rends :t
14 | fin
15 | ve
16 | doubleDraw (1 6)

```



(a) Dessin



(b) Arbre

FIGURE 3 – Exemple fonctions récursives