Université de Technologie de Compiègne

Génie Informatique

Rapport de TP

NF11

Steve LAGACHE & Romain PELLERIN

Chargé de TP : Claude Moulin

Printemps 2016 (P16)

Dernière mise à jour : 7 juin 2016

Table des matières

1 Introduction

Ce rapport est une synthèse d'un ensemble de quatre TP réalisés dans le cadre de l'UV NF11, portant sur le langage Logo (avec ANTLR). En partant d'un embryon de grammaire fourni, il nous fallait, à chaque TP, augmenter les fonctionnalités, notamment en améliorant la grammaire mais aussi le code Java. Le but final de ces TP est d'avoir une grammaire complète du langage Logo, permettant à un utilisateur de taper des instructions, des fonctions ou des procédures, qui seront interprétées par la grammaire et qui généreront un tracé.

2 Grammaire

2.1 Catégories

Au travers de tous ces TP, il nous a été demandé d'ajouter un certains nombres d'instructions ou structures (fonctions, procédures, expressions booléennes, etc.) reconnaissables par la grammaire. Avant de commencer à remplir la grammaire, il nous a fallu à chaque fois réfléchir à la catégorie dans laquelle ajouter chaque nouvelle instruction ou structure. Ainsi, nous avons identifié plusieurs catégories (que l'on peut voir dans notre grammaire finale, trouvable dans l'annexe A). Voici les principales, pour voir l'intégralité de nos catégories, se référer à l'annexe :

— **Une expression** : il s'agit tout simplement d'un valeur entière (`Integer`). Cela peut être sous forme d'un nombre directement, d'une variable, d'un appel de fonction (qui retournera donc une valeur), une somme, une multiplication, un nombre aléatoire (`hasard`) etc.

— **Une expression booléenne** : une valeur `True` ou `False`, résultat de la comparaison de deux expressions.

— **Une instruction** : c'est, comme son nom l'indique, une instruction, directement exécutable dans le programme principal ou depuis une fonction ou procédure. Il s'agit d'une action directe telle que tracer un trait (avancer), lever le crayon, tourner, etc. Une instruction est aussi l'appel à une procédure. Enfin, une instruction peut également prendre la forme de :

- Une boucle (`repete` ou `tantque` par exemple)
- Affecter une valeur à une variable
- Une structure à condition booléenne (si / sinon), qui entraîne l'exécution d'autres instructions en fonction du test booléen
- Retourner une valeur (depuis une fonction)

À noter qu'une instruction prend toujours comme paramètre une expression ou un ensemble d'instructions à appeler (cas d'une boucle par exemple). Il y a une exception cependant : l'appel à une procédure peut se faire sans argument.

— **Une fonction/procédure** : il s'agit d'un bloc déclarant une fonction ou procédure. Nous ne faisons pas de distinction entre les deux, une procédure étant simplement une fonction qui ne retourne rien. Une fonction ou procédure fait appel à une liste d'instructions (au moins une). Par ailleurs, une fonction et une procédure acceptent une liste d'arguments (expressions) facultatifs

2.2 Fonctions versus procédures

Comme précisé ci-dessus, dans la grammaire nous ne faisons pas de différence entre un bloc fonction et un bloc procédure, nous n'avons que le type fonction :

Grammaire

```
functions :
        (function)+
;
```

```
function :
        'pour' NAME (USE)* liste_instructions 'fin'
;
```

(USE)* est une liste d'arguments. La différence notable entre fonction et procédure est qu'une fonction retourne quelque chose et doit s'arrêter dès que l'instruction "retourner valeur" est rencontrée. Là réside donc le plus gros challenge qu'il faudra résoudre en Java. Ainsi, nous avons déclaré la structure :

Grammaire

```
'rends' exp #return
```

comme une expression.

2.3 Appel d'une fonction et d'une procédure

En revanche, nous avons fait une distinction entre l'appel d'une procédure et l'appel d'une fonction. Un appel de procédure est considéré comme une instruction simple :

Grammaire

```
instruction :
  NAME exp* #proc
;
```

où NAME est égal à : NAME : [a-z]+ ;.

En revanche, puisqu'une fonction retourne forcément quelque chose, l'appel d'une fonction est considéré comme **étant une expression et non une instruction** :

Grammaire

```
exp :
  NAME exp* #fun
;
```

À chaque fois, les fonctions et procédures acceptent une liste d'arguments facultatifs.

3 Java

3.1 Code pour les instructions (TP2, TP3 et TP4)

Il s'agissait ici uniquement d'*override*[1] les méthodes Java correspondant à chaque ligne de notre grammaire, dans la classe `LogoTreeVisitor.java`. Lorsqu'il s'agissait d'effectuer des tracés, ou tourner (à gauche ou droite), il suffisait d'appeler des méthodes de `Traceur` pré-existantes.

En revanche, pour les instructions plus complexes telles que la somme, la soustraction, la multiplication, la division ou les boucles, il fallait rajouter un peu de logique, comme dans l'exemple ci-dessous :

Java

```java
@Override
public Integer visitMutl(MutlContext ctx) {
        visitChildren(ctx);
        int left = getAttValue(ctx.exp(0));
        int right = getAttValue(ctx.exp(1));
        setAttValue(ctx, ctx.getChild(1).getText().equals("*") ?
                left * right : left / right);
        return 0;
}
```

Enfin, la gestion de l'atom `loop` a été faite avec une `Stack<Integer>` qui contient une liste d'entiers. À chaque itération dans une boucle `repete`, on ajoute un entier incrémenté par rapport au tour d'avant, en partant de 1. La fonction Java `visitLoop` va faire un `pop()` sur cette pile pour récupérer la dernière valeur ajoutée.

3.2 Expressions booléennes (TP4)

Ici, il nous fallait écrire deux méthodes : une qui traitait l'expression booléenne (et donc rendait `True` ou `False`) et une pour la structure (test booléen puis exécution d'une liste d'instructions si le test est positif, sinon exécution d'une autre liste, le *else*).

Java

```java
@Override
public Integer visitBooleanExpr(BooleanExprContext ctx) {
        visitChildren(ctx);
        int left = getAttValue(ctx.exp(0));
        int right = getAttValue(ctx.exp(1));
        boolean result = true;
        String op = ctx.BOOLEAN().getText();
        if(op.equals("<")){
                result = left < right;
        }
        else if(op.equals(">")){
                result = left > right;
```

1. Réécrire

```
        }
        else if(op.equals("==")){
                result = left == right;
        }
        else if(op.equals("!=")){
                result = left != right;
        }
        else if(op.equals( "<=")){
                result = left <= right;
        }
        else if(op.equals(">=")){
                result = left >= right;
        }
        else {
                System.err.println("Erreur booleen");
        }
        setAttValue(ctx, new Boolean(result));
        return 0;
}
@Override
public Integer visitIfElse(IfElseContext ctx) {
        System.out.println("ifElse");
        visit(ctx.booleanExpr());
        Boolean cond = getBooleanValue(ctx.booleanExpr());
        if(cond.booleanValue()){
                visit(ctx.liste_instructions(0);
        }
        else if (ctx.liste_instructions().size() > 1) {
                visit(ctx.liste_instructions(1);
        }
        return 0;
}
```

Le `else if` de la méthode `visitIfElse` permet de s'assurer qu'un bloc `else` a été donné, car il est facultatif dans la grammaire.

3.3 Fonctions et procédures (TP5)

C'est à partir de cette étape que les choses sont devenues plus complexes.

3.3.1 Fonctionnement d'un appel de fonction sur un ordinateur

Sur un ordinateur, lorsque du code C compilé appelle une fonction, un nouveau contexte de variables locales est créé sur la pile (en mémoire centrale) afin de venir accueillir les variables locales à une fonction et la valeur de retour. Nous avons réfléchi de la même façon pour notre projet, afin d'implémenter un appel à une fonction ou une procédure. Nous avons réfléchi dès le début à comment supporter les appels récursifs.

Par ailleurs, nous voulions également refléter au mieux la réalité : c'est-à-dire stopper l'exécution d'une fonction dès qu'une instruction du type `retourner <valeur>` était rencontrée ; le code éventuellement écrit derrière serait ignoré.

3.3.2 Garder en mémoire les déclarations de fonctions et procédures

Comme vu dans notre grammaire, les blocs de déclaration de fonctions et procédures sont les mêmes. Ainsi, nous avons **créé une nouvelle classe** `Function` : c'est une classe classique Java. Elle possède comme attributs d'objet :

— Son nom

— La liste des noms des arguments qu'elle accepte

— La liste des instructions qu'elle doit exécuter (cet attribut est une instance de `Liste_instructionsContext`)

Elle possède par ailleurs deux méthodes :

— Un constructeur qui s'occupe d'initialiser les attributs d'objet

— Une méthode `public Integer execute(LogoTreeVisitor l)` qui permet d'exécuter les instructions, en faisant un `l.visit(instructions);`

Lorsque la grammaire rencontre un bloc déclarant une fonction ou procédure, notre méthode `visitFunction(FunctionContext ctx)` est appelée. C'est ici que nous instancions un objet `Function` en lui passant les bons paramètres. Ce nouvel objet est ensuite ajouté à une liste des fonctions et procédures connues, qui n'est autre qu'un `HashMap<String, Function>` déclaré comme variable d'instance dans la classe `LogoTreeVisitor`. Cet `HashMap` nous permettra par la suite de récupérer la fonction ou procédure lors de son appel effectif plus tard, afin de l'exécuter.

3.3.3 Classe `TableSymboles`

Nous avons décidé de créer une classe héritant de `HashMap<String, Integer>` qui viendrait contenir toutes nos variables affectées grâce à l'instruction `donne "var 123`. Un `HashMap` nous permet de garantir l'unicité des noms de variables et offre aussi de bonnes performances (temps d'accès constant).

3.3.4 `Stack` de variables

Nous avons créé une variable de type :

Java
```
Stack<TableSymboles> stackVars;
```

dans la classe `LogoTreeVisitor` afin d'empiler des contextes de variables comme expliqué au-dessus dans ce rapport. Voici comme notre pile évolue au fil du temps :

— Dans le constructeur `LogoTreeVisitor`, la pile est initialisée avec une première table des symboles :

Java
```
TableSymboles vars = new TableSymboles();
stackVars.add(vars);
```

— Lorsque l'on affecte une valeur à une variable avec l'instruction `donne`, on l'ajoute dans le contexte de variables le plus haut (donc le sommet de la pile) :

Java
```
@Override
public Integer visitDonne(DonneContext ctx) {
        visitChildren(ctx);
```

```
        String var = ctx.VAR().getText().substring(1); //
           Substring permet d'enlever la double quote, en début
            de nom de variable
        stackVars.peek().put(var, getAttValue(ctx.exp()));
        return 0;
}
```

— De manière similaire, dès qu'on cherche à accéder à la valeur d'une variable, dans la méthode `visitVar()`, on regarde dans le contexte de variable de plus haut niveau (donc le sommet de la pile). On retourne 0 si rien n'est trouvé (c'est un choix de notre part pour éviter d'avoir une exception, on aurait très bien pu traiter le problème autrement) :

Java

```
@Override
public Integer visitVar(VarContext ctx) {
    String var = ctx.USE().getText().substring(1); // Substring
        permet d'enlever les deux points (:) en début de nom de
        variable
        setAttValue(ctx, stackVars.peek().containsKey(var) ?
           stackVars.peek().get(var) : 0);
        return 0;
}
```

3.3.5 Appel de fonction ou procédure

Dans notre grammaire, la seule différence est que l'appel d'une fonction est considérée comme une expression (car il y a un retour de valeur) tandis que l'appel de procédure est une instruction. Cela nous a donc obligé à créer deux méthodes Java. Elles sont très similaires au niveau de leur implémentation. Comme on peut le voir dans la grammaire fournie en annexe A, l'appel d'une procédure correspond à la méthode Java `visitProc(ProcContext ctx)` tandis que l'appel d'une fonction correspond à la fonction Java `visitFun(FunContext ctx)`.

Puisque les deux implémentations sont similaires, voyons celle de l'appel d'une fonction. La fonction est directement expliquée en commentaires :

Java

```
private static final String RETURN_VAL = "return";

...

@Override
public Integer visitFun(FunContext ctx) {
TableSymboles vars = new TableSymboles(); // On crée un nouveau
   contexte de variables pour cette fonction

String funcName = ctx.NAME().getText(); // On récupère le nom de la
    fonction que l'on appelle
Function f = funcProc.get(funcName); // On récupère la fonction cré
   ée lors de sa déclaration

if (ctx.exp() != null) { // S'il y a des arguments...
```

```
    ArrayList<String> args = f.getArgs(); // On récupère le nom de
       ses arguments

    for (int i = 0; i < ctx.exp().size(); i++) { // On itère sur
       les arguments qui lui ont été donnés lors de l'appel
        visit(ctx.exp(i)); // Puisque c'est des expressions, on les
            visite
        vars.put(args.get(i).substring(1), getAttValue(ctx.exp(i)))
           ; // On ajoute la variable (son nom et sa valeur) dans
           notre contexte actuel
        // Le substring permet d'enlever le ':' présent au début du
            nom de variable
    }
}

stackVars.push(vars); // On ajoute ce nouveau contexte de variables
    en haut de notre pile

f.execute(this); // Execution
Integer returnValue = stackVars.peek().get(RETURN_VAL);
if (returnValue != null) { // on vérifie la présence d'une variable
    de retour
        System.out.println("returnValue found");
        setAttValue(ctx, returnValue); // Retourne la valeur
}
else { // Normalement, avec notre grammaire, ce cas est impossible
        System.out.println("returnValue NOT found");
}

stackVars.pop(); // On détruit le contexte de variables propre à la
    fonction
return 0;
}
```

L'implémentation de la fonction d'appel d'une procédure est la même sauf qu'on ne traite pas de variable de retour.

3.3.6 Arrêt d'exécution en cas de retour de valeur

L'implémentation vue précédemment autorise les appels récursifs sans problème.

La dernière étape qu'il convenait de faire et d'arrêter l'exécution d'une fonction dès qu'une instruction `rends x` était rencontrée. En fait, cela se fait de manière très simple. Voici tout d'abord la fonction qui visite une instruction `rends` :

Java

```
@Override
public Integer visitReturn(ReturnContext ctx) {
        visitChildren(ctx);
        Integer previous = stackVars.peek().put(RETURN_VAL,
           getAttValue(ctx.exp()));
        if (previous != null) { // Cas normalement impossible avec
           notre implémentation
```

```
                stackVars.peek().put(RETURN_VAL, previous);
                return -2;
        }
        return 0;
}
```

L'exécution d'une fonction revient à visiter ses enfants. Donc la méthode `visitListe_instructions (Liste_instructionsContext ctx)` sera appelée. C'est à l'intérieur de cette méthode qu'il faut vérifier, avant chaque instruction, que le contexte actuel de variable ne contient pas déjà de variable de retour.

Java

```
@Override
public Integer visitListe_instructions(Liste_instructionsContext
   ctx) {
        int ret;
        for (InstructionContext ic : ctx.instruction()) {
                ret = visit(ic);
                if (ret == -1 || ic instanceof ReturnContext) {
            // La liste d'instructions contenait un return, on
               stoppe
                        return -1;
                }
        }
        return 0;
}
```

Ici, le détail qui change tout est que nous retournons -1. C'est une valeur qu'il faut donc ensuite vérifier dans toutes les instructions ou structures faisant appel à une liste d'instruction. En effet, dans un `tantque`, si l'exécution de la liste d'instructions est interrompue car il y a un `rends`, on ne veut pas poursuivre l'itération suivante de la boucle. C'est un exemple simple mais il faut savoir qu'il y a plusieurs instructions qui comportent des listes d'instructions, notamment :

- `repete'atom '['liste_instructions ']'#repete'`
- `'si' booleanExpr '['liste_instructions ']''['liste_instructions ']'? # ifElse`
- `'tantque'booleanExpr '['liste_instructions ']'# tantQue`

Ainsi, à chaque fois, nous avons dû ajouter des vérifications sur la valeur de retour de la visite des instructions, avant de continuer, comme dans l'exemple ci-dessous :

Java

```
@Override
public Integer visitTantQue(TantQueContext ctx) {
        visit(ctx.booleanExpr());
        while(getBooleanValue(ctx.booleanExpr()).booleanValue()){
                if (visit(ctx.liste_instructions()) != 0)
                        return -1;
                visit(ctx.booleanExpr());
        }
        return 0;
}
```

La valeur de retour -1 doit impérativement être transmise également par ces méthodes Java pour être ainsi interceptées sur plusieurs sous-niveaux. En effet, la méthode `visitListe_instructions` vérifie également que chaque instruction ne renvoie pas -1.

Toutes ces explications quant à la gestion de l'arrêt d'une fonction peuvent paraître complexes à comprendre. Elles sont surtout difficiles à traduire dans un rapport. En cas de doutes, nous pourrons fournir de plus amples détails lors de la soutenance.

Inconvénient

Notre variable de retour est enregistrée dans notre contexte de variables sous le nom "`return`" (comme on peut le voir dans l'extrait de code page 9 de ce rapport). Cela empêche donc l'utilisateur de nommer une de ses variables ainsi. C'est le seul inconvénient.

4 Conclusion et screenshots

En somme, nous avons implémenté toutes les fonctionnalités requises, en supportant même l'arrêt d'exécution avec avoir rencontré l'instruction `rends`. Voici un exemple de code exécuté avec le résultat visuel. On peut voir l'utilisation d'une fonction et d'une procédure.

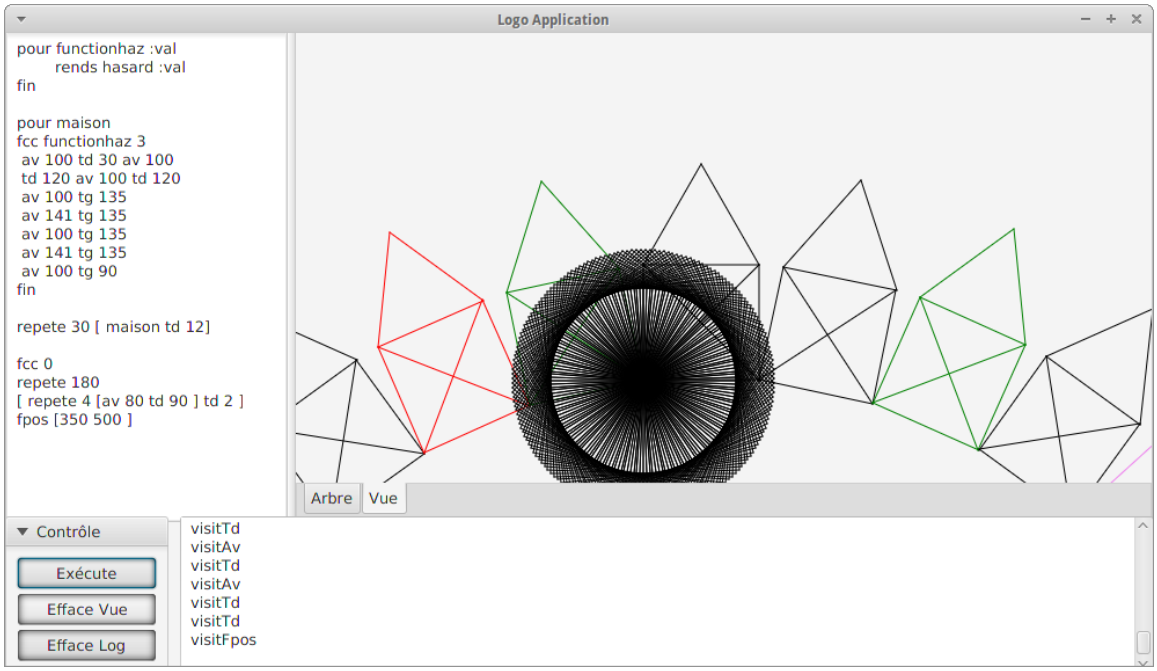

Figure 4.1 – Exécution de code

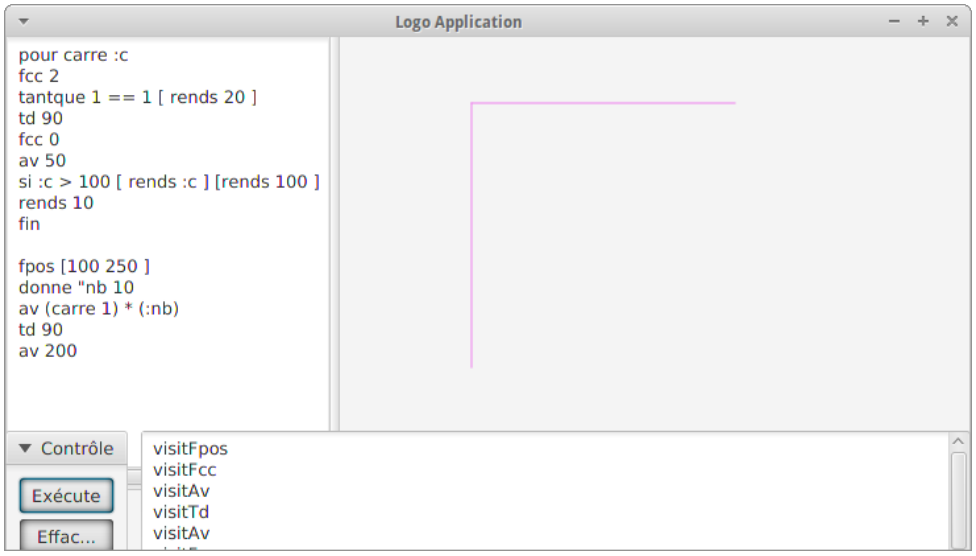

Figure 4.2 – Autre exemple

Sur ce deuxième exemple, on peut voir que ce qui suit le premier `rends` est ignoré. `fcc 2` met la couleur en violet, `fcc 0` met la couleur en noir. Cela fonctionne comme attendu.

Enfin, voici le même programmé, simplifié, et son arbre associé :

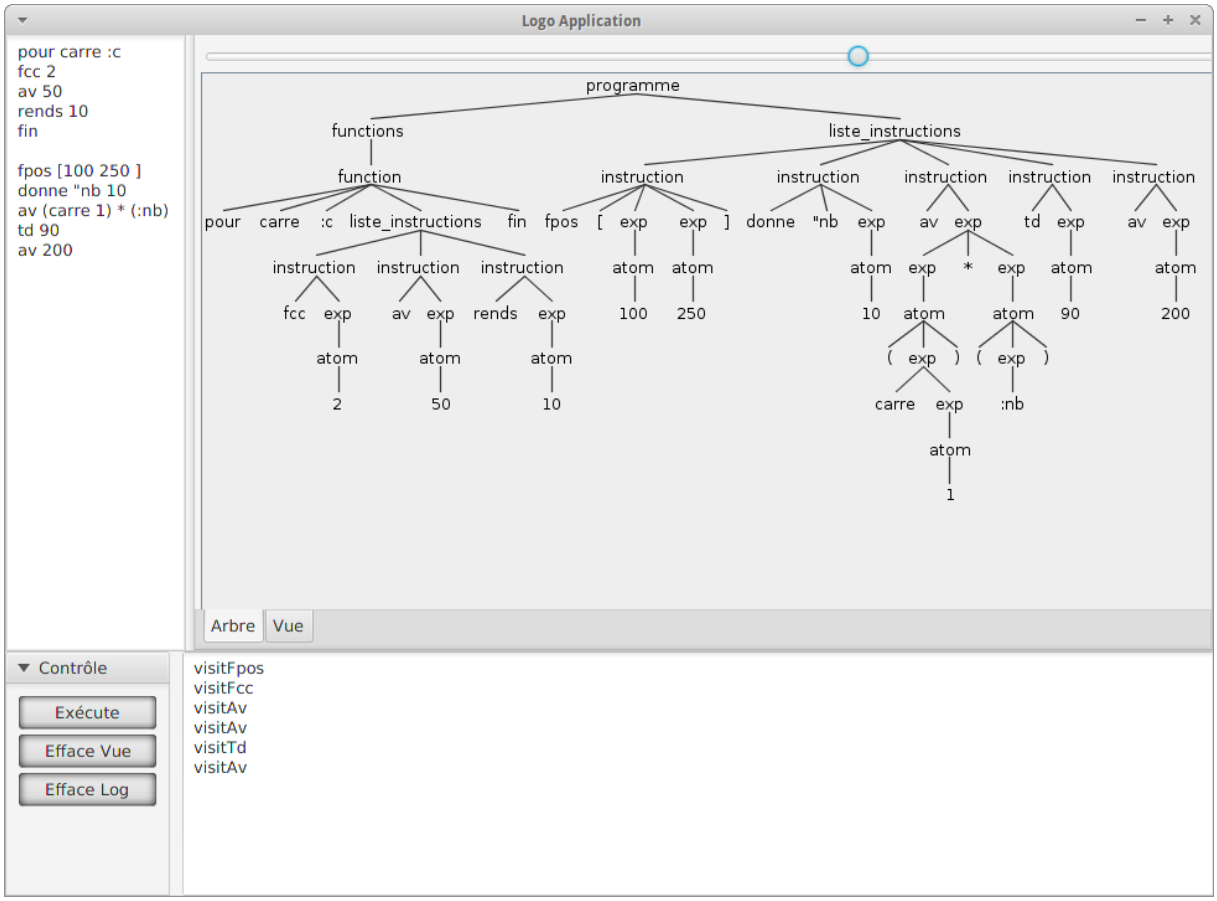

Figure 4.3 – Arbre généré

Appendices

A Grammaire finale

Java

```
grammar Logo;

@header {
  package logoparsing;
}

INT : '0' | [1-9][0-9]* ;
WS : [ \t\r\n]+ -> skip ;
BOOLEAN : '<' | '>'| '<=' | '>=' | '==' | '!=' ;
VAR : '"'[a-z]+ ;
USE : ':'[a-z]+ ;
NAME : [a-z]+ ;

programme : functions? liste_instructions
;
liste_instructions :
  (instruction)+
;
functions :
        (function)+
;

function :
        'pour' NAME (USE)* liste_instructions 'fin'
;
instruction :
    'av' exp # av
  | 'td' exp # td
  | 'tg' exp # tg
  | 'lc' # lc
  | 'bc' # bc
  | 've' # ve
  | 're' exp # re
  | 'fpos' '[' exp exp ']' # fpos
  | 'fcc' exp # fcc
  | 'repete' atom '[' liste_instructions ']' #repete
  | 'donne' VAR exp # donne
  | 'si' booleanExpr '[' liste_instructions ']' '['
     liste_instructions ']'? # ifElse
  | 'tantque' booleanExpr '[' liste_instructions ']' # tantQue
  | NAME exp* #proc
  | 'rends' exp #return
;
```

```
booleanExpr : exp BOOLEAN exp;

exp :
          exp ('*' | '/') exp # mutl
        | exp ('+' | '-') exp # sum
        | 'hasard' exp # haz
        | atom # aroule
        | USE # var
        | NAME exp* #fun
;

atom :
        INT # int
        | '(' exp ')' # parenthese
        | 'loop' # loop
;
```