Coqueux Damien
Gomez Guillaume

# NF11

# *Génération d'analyseur lexical et syntaxique*

## 1) La grammaire

Elle est à la base de tout notre projet. La grammaire définit notre langage qui est un ensemble d'instructions (sous le chef de la règle intitulée "liste_instructions"). Ces instructions représentent différentes composantes de la redéfinition du langage "Logo". On y retrouvera, les actions permettant de remplir les fonctionnalités de dessin, les conditionnelles, la déclaration et l'affectation de variable, les boucles à itérations définies et indéfinies, et enfin les fonctions et les procédures. Pour faciliter l'analyse du code écrit par l'utilisateur, notre grammaire oblige que les déclarations des fonctions (ou procédures) se fassent en début de programme.

Toutes les expressions arithmétiques se retrouvent sous un "package" de règles intitulé "expr" dans notre grammaire. Ainsi, toute chaîne de lexèmes qui signifie une valeur numérique est englobée dans cette package.

Le groupe "condition" permet la gestion d'instructions conditionnelles au sein d'un même structure. Chacune des conditionnelles se fait entre deux expressions arithmétiques. Ainsi on pourra réaliser des conditions avec des valeurs triviales ou beaucoup plus élaborés.

Par ailleurs, nous avons créé une règle "bloc" qui permet de structurer une suite d'instructions. L'apport de cette règle est qu'il facilite l'écriture de la règle des structures conditionnelles (si ….) et de boucle (repete … / tantque …) . De plus, la lecture par l'arbre de dérivation est plus aisée.

La règle "variable" est triviale : elle précise qu'une variable est une suite de caractères. Ensuite, elle est utilisée en tant que déclaration à travers la règle "instruction" pour pouvoir le utiliser cette instruction dans n'importe quel endroit du code. L'utilisation de la variable se fait sous la forme d'alternative à la règle "expr". Ainsi, chaque variable peut être intégrée dans toutes les expressions arithmétiques.

Nous avons décidé de décomposer la déclaration et l'exécution des fonctions et des procédures, puisque syntaxiquement différentes. De plus, pour les fonctions et les procédures, nous avons choisi pour chacun des deux concepts de distinguer la règle "XXX_execution" de la règle "XXX_declaration". En effet, l'interpréteur ne saurait pas s'il doit exécuter du code (exécution de la fonction/procédure) ou non (déclaration de celle ci).

Cela nous fait donc 4 règles pour fonctions et procédures. Nous avons laissé la distinction entre l'exécution d'une fonction et d'une procédure puisque si la première retourne forcement une valeur, la seconde non. Cette différence à une répercussion sur la grammaire: la procédure est une alternative de la règle "instruction", et la fonction est une alternative de la règle "expr" car le retour de la fonction doit pouvoir s'intégrer dans un calcul (et par conséquence dans une affection de variable).

## 2) Architecture globale

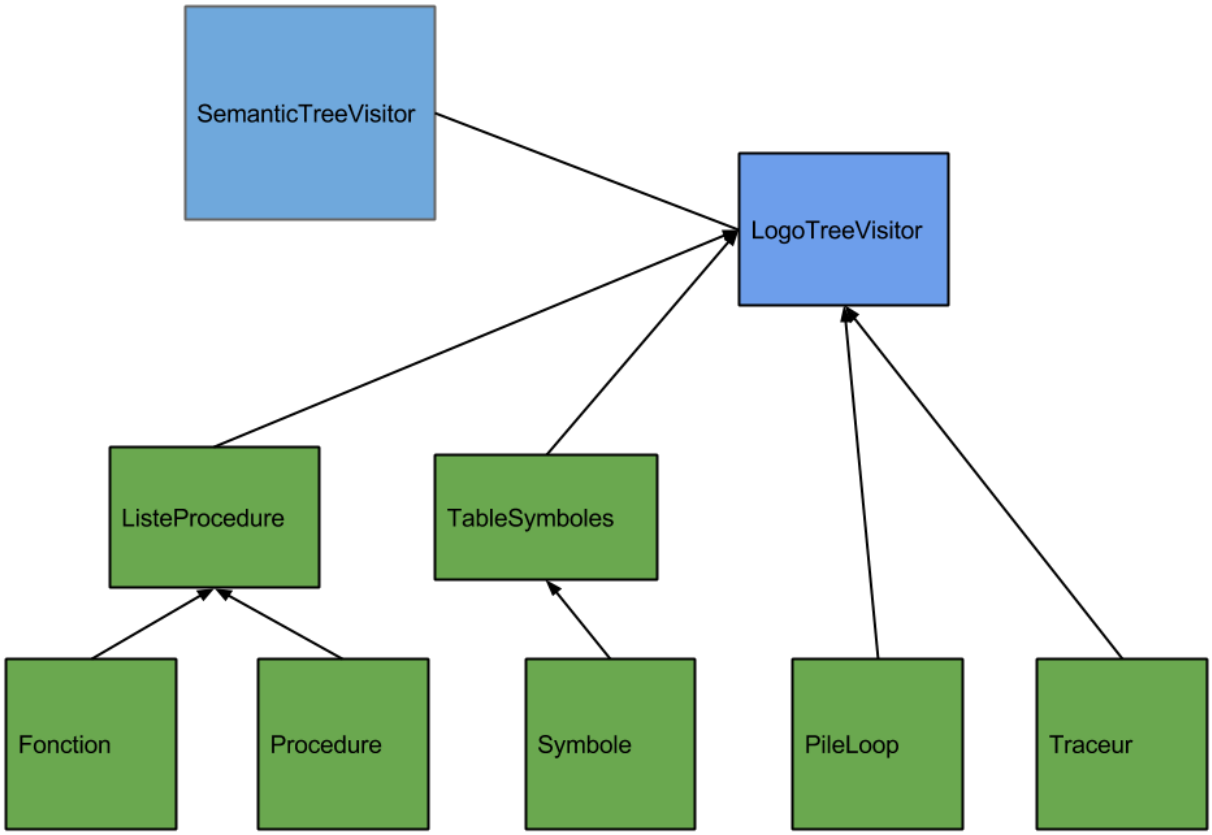



Figure 1 : Schéma de l'architecture globale du compilateur

SemanticTreeVisitor : Parcourt l'arbre de dérivation pour faire l'analyse sémantique du code écrit.

LogoTreeVisitor : Parcourt l'arbre de dérivation pour faire en vu d'exécuter le code.

Traceur : Classe qui réalise les fonctions de dessin de logo.

PileLoop : Classe pour la gestion des variables "Loop"

Symbole : Gère une liste de variables déclarées dans le code.

TableSymbole : Stocke l'ensemble des variables déclarés en prenant en compte la portée des variables.

Procedure : enregistre les informations nécessaires pour l'exécution d'une procédure.

Fonction : idem que Procedure, avec l'ajout d'une valeur de retour.

ListeProcedure : Stocke l'ensemble des fonctions et des procédures déclarées dans le code.

## 2. Points particuliers de la création

### 2.1. Gestion des variables

Contrairement à d'autres langages comme le C par exemple, il n'y a pas de déclaration de variables à proprement dit. L'utilisateur est libre de rajouter une variable au milieu du programme principal, tant qu'une instruction ne nécessite pas de variables. A sa création donc, la variable est automatiquement initialisée : la grammaire l'impose. En effet, nous n'avons qu'une seule règle qui traite les variables, et celle-ci permet la déclaration et la modification de la valeur d'une variable. A noter que nos variables ne sont que d'un seul type : *Entier (Integer pour Java)*.

L'initialisation d'une variable se fait au moyen de la syntaxe suivante:

```
donne "<nom_var> <valeur>
```

#### 2.1.2. Portée et visibilité

La portée et la visibilité des variables dépend du contexte (bloc) dans laquelle celle-ci fut créée. En d'autres termes, une variable écrite au sein du programme principal n'aura pas la même portée qu'une variable déclarée dans une procédure ou une fonction (= variable locale). Ainsi une variable locale n'existe que dans le contexte de la procédure/fonction qui lui est associé. Nous n'avons pas à proprement parler de variables globales avec notre travail.

Dans notre langage, nous différencions les portée et la visibilité des variables seulement entre programme principal et l'ensemble procedure-fonction. Mais nous aurions pu implementer comme en *java* ou *c++* la limitation de la visibilité à chaque construction de bloc (conditionnelle, boucle).

Nous avons dit plus haut que les variables globales ne sont visibles que dans le programme principal et pas dans les fonctions/procedures. Bien qu'il soit tout fait possible en lecture de récupérer une variable globale avec notre architecture, le probleme se poserait lors de création ou modification d'une variable dans un contexte local.

En effet, si le programmeur crée une fonction *fonc1* avec une variable locale *var*, et que le programme principal contient aussi une variable *var.* Puisque dans sa fonction il redéfinit la variable local *var* :comment savons nous si il change la valeur de variable locale ou celle de la variable globale! Pour éviter ce problème, nous avons renoncé à cette possibilité pour permettre au programmeur d'utiliser des variables locale ayant le même nom qu'une des variables globales. Si celui souhaite posséder la valeur d'une variable globale, il devra la passer en paramètre.

Pour corriger ce problème , il aurait fallu changer la grammaire et proposer une règle propre à la déclaration et une autre à l'affection.

*Exemple d'erreur possible si on avait élargi le contexte des variables globales :*

```
pour proc
av varGlobale //lecture de la var globale
donne varGlobale 78  //est ce qu'on change la variable globale ou on crée une nouvelle variable?
fin

donne "varGlobale 45
proc[]
```

### 2.1.3. Utilisation et cas d'erreur

L'utilisation d'une variable se fait au moyen de la syntaxe `:<nom_var>`
Bien entendu, il est possible d'insérer la variable dans une expression arithmétique.

Exemple d'utilisation :

```
donne "var1 100              // Affectation var1 à 100
av :var1                     // Avance de 50
tg :var1 + 3 * 20            // Tourne à gauche de 160°
```

Un type d'erreur a été identifié, il s'agit de l'utilisation d'une variable qui n'a jamais été initialisée.

Exemple d'erreur 1 :

```
donne "var1 100          // Affectation var1 à 100
av :var1                 // Avance de 100
av :var2                 // Erreur : variable "var2" non définie
```

Exemple d'erreur 2 :

```
Pour proc
      donne "var2 50
      av :var1 + :var2
fin


donne "var1 100
proc[]                    // Erreur : variable "var1" non définie
av :var2                 // Erreur : variable "var2" non définie
```

### 2.1.4. Implémentation

Pour implémenter la gestion d'une variable, nous avons créé une classe java *Symbole* qui est tout simplement composée d'un tableau associatif (HashMap) qui associe une chaîne de caractères (nom de la variable) à un entier (valeur de la variable). Cette classe offre deux méthodes publiques pour manipuler les variables :

```
public int getSymbole(String symbole) // lecture d'une variable passée
en paramètre

public void ajouterSymbole(String symbole, int valeur)
 /* affectation
d'une variable, si elle existe déjà sa valeur sera modifiée*/
```

Afin de gérer la portée des variables, nous avons intégré la classe *Symbole* dans une autre classe composantes qui se nomme *TableSymbole*. Elle contient un pile (Stack) de Symbole. Chaque niveau de la pile correspond à un contexte. Ainsi le premier niveau de la pile contiendra les variables du programme principal, puis, dès que l'on appelle une fonction ou une procédure, on se place dans un nouveau contexte, et les variables locales à cette fonction/procédure se trouve au niveau supérieur de la pile de *TableSymbole*

Aux méthodes de *Symbole* s'ajoutent les méthode suivantes:

```
 public void creerNouveauSymbole() // création d'un nouveau contexte
dans la pile
  public void depilerSymbolesProcedure()  // accesseur dans du dernier
élément de la pile
```

De plus, la classe possède un attribut *error* pour gérer les erreurs sur des appels de variables indéfinis.

## 2.2. Procédures et fonctions

### 2.2.1. Déclaration

La déclaration d'une procédure ou d'une fonction se ressemblent fortement syntaxiquement parlant, a ceci près que la fonction possède obligatoirement l'instruction *rends* qui signifie le retour de la fonction. Cependant, les méthodes de visite de l'arbre LogoTreeVisitor sont semblables.

Syntaxe d'une procédure :

```
Pour <nom_procedure> :<param1> :<param2> … :<paramN>
     // Instructions
fin
```

Déclaration d'une fonction :

```
Pour <nom_fonction> :<param1> :<param2> … :<paramN>
     // Instructions
     rends <expression>
fin
```

### 2.2.2. Instruction de retour (rends)

L'instruction de retour d'une fonction est particulière, elle n'est pas d'ailleurs traitée au même titre que les autres instructions. En effet, notre grammaire nous oblige à placer cette

instruction à la toute fin d'une déclaration uniquement, il est impossible qu'elle soit placée en dans le corps de cette déclaration (dans un bloc conditionnel par exemple). Cette spécificité de placement, nous devons bien le reconnaître, n'est pas idéale dans certains cas, comme celui tout juste cité. Ce problème vient de notre grammaire qui reconnaît les fonctions grâce à cette instruction.

De plus, il n'est pas si simple d'arrêter l'exécution d'un bloc de code et de rendre la main au contexte appelant, d'autant que la structure actuelle de nos méthodes et de l'exécution des instructions n'a pas été pensée pour s'accommoder à un arrêt en cours d'exécution. Par manque de solutions techniques, nous avons décidé de laisser ce mode de fonctionnement.

### 2.2.3. Appel d'une procédure/fonction

L'appel d'une fonction se fait de manière assez intuitive, simplement en précisant le nom de la procédure/fonction, et en y accolant une paire de crochets qui contiennent éventuellement les paramètres :

```
<nom_fonction/procedure> [ <param1> <param2> ... <paramN> ]
```

Ambiguîté de l'appel d'une fonction/procèdure:

Comme on peut le voir plus haut, l'appel d'une fonction et d'une procédure est semblable. Hors, nous avons choisi de définir dans la grammaire :

- qu'une fonction est une expression (comme toutes les expressions arithmétiques)
- qu'une procédure est une instruction.

Ainsi, on évitera qu'une procédure soit appelée avec une instruction qui renverrait une erreur (cette erreur est traitée et expliquée plus précisément dans la rubrique "Traitement des erreurs").

Cependant, dans le cas où une fonction est utilisée comme une procédure, le cas est moins problématique. En effet, comme "procedure_execution" et "fonction_execution" ont la même syntaxe, il n'y a pas d'erreur grammaticale. Ainsi, si on utilise une fonction comme une 'instruction, LogoTreeVisitor l'interprétera comme une procédure, c'est à dire en visitant les instruction de la fonction sans visiter l'instruction *rends*. La valeur de retour de la fonction sera simplement perdue.

Justification de la présence des crochets:

La première version développée ne contenait pas de parenthèses, ce qui dans certains cas créait des ambiguïtés. Prenons une fonction *inverse* qui renvoie le double d'un entier passé en paramètre et utilisons cette fonction dans une expression arithmétique, ce qui sans parenthèse donnerait par exemple :

```
av carre 10 + 100
```

L'ambiguïté est ici de savoir si on doit avancer du carré de 10, ajouté de 100, ou bien si on doit avancer du carré de (100 + 10), à savoir 110². Bien qu'il soit possible de définir une règle pour que les expressions de ce type soient toujours interprétées de la même façon, il n'en reste pas moins que l'ambiguïté est toujours présente pour l'utilisateur. De ce fait, il a été décidé que l'utilisation d'une fonction se ferait via l'usage de crochets.

### 2.2.4. Implémentation

Pour implémenter les procédures et les fonctions, nous avons créé une classe *Procedure*. Cette classe est notamment composée :

- d'un nom
- d'une liste de variables qui représente les paramètres de la fonction,
- d'une liste d'instructions qui compose la procédure,

Pour les fonctions, nous avons développé une classe *Fonction* qui hérite de *Procedure*. Elle permet juste de différencier intelligemment les procédures et fonctions et d'utiliser le polymorphisme qui permet de retourner une valeur dans le cas des fonctions (ce qui ne sera pas la cas dans les procédures)

Pour le stockage, nous utilisons une seconde classe composante qui se nomme *ListeProcedure* qui stockent les fonctions et procédures dans un même tableau, déclaré dans le programme principal. Ainsi, lors de la définition d'une procédure (ou fonction), on stocke cette nouvelle procédure (ou fonction), on récupère les informations primordiales en vue de les sauvegarder dans *ListeProcedure.*

ListeProcedure ne gère pas les contextes (à quel niveau de récursivité nous sommes par exemple). Cette problématique est déjà centralisée dans la classe *ListeSymbole*. Ainsi, lors d'un appel d'une fonction ou d'une procédure; un nouveau contexte est créé, puis supprimé en fin d'appel.

Dans le cas récursif justement, notre système fonctionne, puisque décentralisée. Pour des raisons de pertinence des données, nous limitons en testant si la pile de contextes est trop grande. En effet, lors de nos tests, la fonction *factorielle* donnait des résultats négatifs pour des valeurs supérieures à n = 12

```
fact[ 5 ] // retourne 120
fact [ 17 ] //retourne -1960358400 ainsi qu'un message d'erreur
```

## 2.1. Instructions spéciales

Il existe quelques instructions particulières dans notre grammaire. C'est le cas notamment de l'instruction *Rends* qui a déjà été évoquée plus haut (nous n'en reparlerons pas ici). L'instruction *Stop* n'a pas été implémentée faute de temps.

En revanche, l'instruction *loop* nous a été demandé. Cette instruction dans le cadre d'une boucle *repete* joue le rôle de compteur au sein de la boucle, qui s'incrémente à la fin de parcours d'une boucle. Pour implémenter cette fonctionnalité, on utilise la classe *PileLoop* qui contient une pile de *int.* La structure de pile a été préférée dans les cas où nous avons plusieurs boucles définies imbriquées.

Comme pour les variables, dès que l'on s'enfonce un peu plus au coeur du bloc, un nouveau "contexte" est matérialisée par l'empilement d'un nouveau *loop* dès que l'on parcourt une nouvelle boucle imbriquée. A la fin de chaque itération d'une boucle *repete*, on incrémente la valeur du *loop* en tête de la pile. Lorsque la dernière itération de la boucle est terminée, on dépile ce *loop* et son contexte.

Exemple d'utilisation de loop :

```
Exemple 1:
// Boucle équivalente à av 10 + 20 + 30 + 40 + 50
Repete 5
[
      av 10 * loop
]

Exemple 2:
//construction d'un premier loop : empilement niveau 1
Repete 4
[
      //constuction d'un second loop : empilement niveau 2
      Repete 8
      [
            av 20
            fcc loop      //utilisation du second loop
      ]
      //destruction du second loop :depilement niveau 2
```

```
 td 45 * (loop + 1) // utilisation du loop en tete de pile : niveau 1
]
//destruction du premier loop : depilement niveau 1
```

## 3. Traitement des erreurs

### 3.1. Startégie de gestion des erreurs

Pour qu'un programme s'exécute dans l'interpréteur LOGO, il doit passer par une étape de validation avant une étape d'exécution. On commence par la validation pour vérifier et s'assurer que tout le programme est valide syntaxiquement, si cette étape s'est déroulée avec succès, on passe à l'étape de l'exécution. Sinon, l'exécution ne se réalise pas : pas de vue graphique.

Lorsque des erreurs sont détectées dans l'étape de validation, celles-ci doivent être traitées par l'utilisateur avant de pouvoir exécuter le programme. Nous avons choisi qu'à la moindre erreur, l'exécution du programme est annulée, et le message d'erreur est transmis dans la console de l'application graphique de telle sorte que l'utilisateur soit informée de sa faute pour la corriger en conséquence. Lorsqu'une erreur survient lors de la phase de validation, le résultat affiché est le suivant :

```
************************************************
Erreur la variable "var" n'existe pas
!----- ERREUR ! Fin d'analyse -----!
************************************************
```

Néanmoins, nous avons aussi considéré qu'il peut y avoir plusieurs erreurs. Si c'est le cas, la détection d'une erreur n'est pas bloquant, et on les affiche toutes lors d'une même validation par souci d'ergonomie, afin que l'utilisateur puisse corriger l'ensemble des erreurs et ainsi exécuter le programme avec succès. Lorsque plusieurs erreurs surviennent lors de la phase de validation, le résultat affiché est le suivant.

```
************************************************************************
Erreur --> le nombre de paramètres ne correspondent pas pour la
procedure "fonc"
nombre de param attendus : 1
nombre de param lus : 2
Erreur la variable "var" n'existe pas
!----- ERREUR ! Fin d'analyse -----!
```

**********

### 3.2. Types d'erreurs

Dès que la moindre erreur a été détectée, comme précisé ci-avant, on annule l'exécution du programme et on affiche dans le Log le type d'erreur qui est apparu. Voici ci-dessous les différents types d'erreurs que nous avons identifiés.

#### 3.2.1. L'utilisation d'une variable non affectée

Le premier type d'erreurs est d'utiliser dans le programme principal une variable qui n'a jamais été affectée. Le code suivant contient une erreur de ce type.

```
donne "b 49      //affectation de variable
av :a           // erreur ! la variable n'existe pas
```

A l'exécution de ce code, on affiche le log suivant :

```
**************************************************
Erreur la variable "a" n'existe pas
!----- ERREUR ! Fin d'analyse -----!
**************************************************
```

#### 3.2.2. L'appel à une procédure ou une fonction non déclarée

Un second type d'erreur identifiable est l'appel à une procédure/fonction qui n'a pas été déclarée. Le code suivant contient une erreur de ce type :

```
donne "a 10

 proc[]              //On appel une procédure "proc" qui
                          //n'est pas été déclarée
```

A l'exécution de ce code, on affiche le log suivant :

```
************************************************
Impossible de trouver 'proc'
!----- ERREUR ! Fin d'analyse -----!
************************************************
```

### 3.2.3. L'utilisation d'une procédure ou d'une fonction avec une arité incorrecte

On a aussi considéré le cas dans lequel on utilise une procédure ou une fonction en lui envoyant un nombre de paramètres (l'arité d'une fonction) différent de la déclaration de ladite procédure ou ladite fonction. L'exemple ci-dessous représente un cas dans lequel on peut trouver ce type d'erreur :

```
//On déclare une procédure proc
pour proc :arg
 av :arg                    //avec un seul paramètre "arg"
fin
donne "a 10
proc []  // on veut appeler proc sans paramètre
```

A l'exécution de ce code, on affiche le log suivant :

```
************************************************************
Erreur --> le nombre de parametres ne correspondent pas pour la
procedure "proc"
nombre de param attendus : 1
nombre de param lus : 0
!----- ERREUR ! Fin d'analyse -----!
************************************************************
```

### 3.2.4. L'appel d'une procédure à la place d'une fonction

Pour rappel, la différence entre une procédure et une fonction est qu'une fonction retourne une valeur entière par l'instruction de retour *rends*. De cette manière, on peut utiliser un appel de fonction dans n'importe quelle expression arithmétique. Mais si on le fait avec une procédure, celle-ci sera grammaticalement considérée comme une instruction, qui ne peut pas

s'inclure dans une expression arithmétique. Nous avons choisi de traiter ce genre de cas. Ainsi, l'appel à une procédure là où une fonction est attendu génère une erreur.

```
Pour proc             //On déclare une procédure proc
 Repete 4[av 50 td 90]     //qui ne retourne rien
fin

av proc[]             //On souhaite avancer de la valeur
                           // retourner par proc
```

A l'exécution de ce code, on affiche le log suivant :

```
**************************************************************
Erreur impossible de convertir une Procedure en Fonction
!----- ERREUR ! Fin d'analyse -----!
**************************************************************
```

On précise ainsi qu'on a trouvé une procédure à la place d'une fonction.

### 3.2.5. Loop en dehors d'une boucle

Quant on se trouve à l'intérieur d'une instruction **repete,** on a accès à une variable appelée **loop** qui renvoie le nombre de fois que la boucle a été exécutée. Cette instruction est propre au contexte de la boucle, la présence de cette instruction à l'extérieur d'une boucle n'a pas de sens. Dès lors, nous avons considéré qu'il s'agissait d'une erreur et ainsi l'avons traiter comme telle. Elle est d'ailleurs facile à détecter : si loop se trouve dans le contexte d'une boucle, alors PileLoop contient une pile non vide car utilisé par le bloc repete. En dehors, le visiteur est confronté à une pile vide. Ci-dessous le code correspondant à cette erreur.

```
donne "a 10
Tant que :a < 20
[
 Repete 10[av 5 * loop td 30]  // Utilisation de loop correct
 donne "a :a + 1
]
av loop                        //On utilise la variable loop en
                               //dehors de la boucle -> Erreur
```

A l'exécution de ce code, on affiche le log suivant :

```
**************************************************************
Erreur loop ne pas être appelée dans ces conditions
**************************************************************
```

### 3.3. Implémentation

Pour réaliser une validation du code avant son exécution nous avons créé une classe *LogoSemanticVisitor* semblable à la classe *LogoTreeVisitor*, qui parcourt les noeuds du programme. Cette classe se contente de vérifier les erreurs vues plus haut. Cependant, en aucun cas les noeuds ne sont visités car cela pourrait générer des erreurs de compilation par AntLR.

Dans cette classe, si une erreur est detectée, l'attribut *error* passe à false. Dans le cas d'une alternative, les deux cas (if et else) sont testés, si *error* est faux, alors aucun des noeuds de l'arbre de dérivation ne sera parcouru.

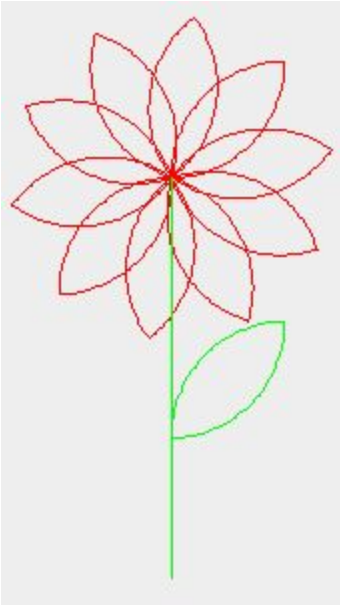
## 4 Exemples :

Premiere exemple : une fleur

```
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
av 100
fcc 1
fleur []
fcc 2
```

```
re 130
petale []
re 70
fin

ve
plante []
```

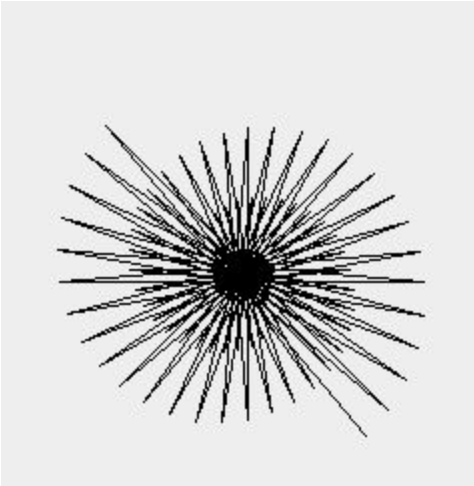
Voici le résultat d'une telle fonction :



Second exemple: une spirale

```
pour spir :cote :angle
av :cote
td :angle
    si :cote <= 200
       [spir [:cote +2 :angle]]
fin
ve
spir [2 185]
```

Cela donne :

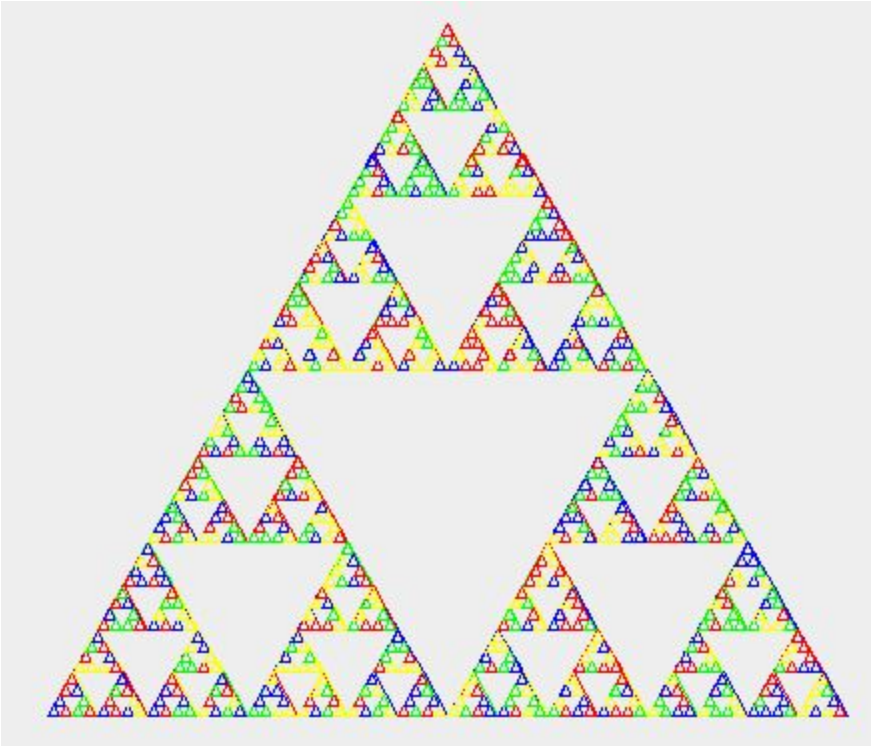


Troisième exemple:

```
pour triangle :n :t
donne "bool 1
si :n < 4 [donne "bool 0]
si :bool = 1
[
fcc hasard 5
repete 3 [av :n td 120]
donne "t :t + 1
triangle [:n / 2 :t]
lc av :n / 2 bc
triangle [:n / 2 :t]
lc td 120 av :n / 2 tg 120
bc
triangle [:n / 2 :t]
lc td 240 av :n / 2 td 120 bc
]
fin


ve
lc
fpos [400 400]
bc
```

```
td 30
triangle [300 0]
```

Résultat :



## 5 Conclusion :

Ce projet nous a permis de consolider réellement nos connaissances en matière de compilation et d'analyse d'un code source dans un langage quelconque, à partir d'une grammaire qui le définit. Nous avons mieux compris tous les problématiques autour de chaque élément qui constitue un langage informatique simple : les boucles, les fonctions, les expressions arithmétiques.

Nous regrettons un peu que le temps nous ait manqué pour implémenter le reste des fonctionnalités que propose le langage LOGO : les mots-clés ***stop***, ***rends*** à n'importe quel niveau de la déclaration d'une fonction notamment.