Théorie des Langages

AntLR - Générateur d'analyseurs

Claude Moulin

Université de Technologie de Compiègne

NF11

AntLR

- ANTLR : ANother Tool for Language Recognition
- URL : `http://www.antlr.org`.
- Auteur : Terence Parr (Université de San Francisco)
- Il fournit un parser qui interagit avec le lexer, fournit un rapport d'erreur assez explicite et construit des arbres syntaxiques.

Objectif

- AntLR est un outil de génération automatique d'analyseur permettant de reconnaître les phrases d'un programme écrit dans un langage donné.
- L'analyseur est généré à partir d'un fichier grammaire contenant les règles définissant le langage.
  - Code généré : classes en Java, C# ou C++, Python, etc.
- AntLR définit un Méta Langage utilisé pour écrire le fichier grammaire.

Sommaire

Génération du code : Windows

- On crée un fichier batch qui appelle le générateur d'AntLR et permet de générer le code à partir d'un fichier grammaire g4
- antlr-generate.bat

```
@echo off
set CLASSPATH=lib/antlr-4.9.1-complete.jar;
java org.antlr.v4.Tool -visitor
          -o src/logoparsing  grammars/Logo.g4
pause
```

Syntaxe :

```
java org.antlr.v4.Tool -visitor
              -o <output-folder>
                  <grammaire.g4> ;
```

Génération du code : Linux

- antlr-generat.sh

```
export SAVECLASSPATH=$CLASSPATH
export CLASSPATH=lib/antlr-4.9.1-complete.jar

java org.antlr.v4.Tool -visitor
              -o src/logoparsing grammar/Logo.g4
export CLASSPATH=$SAVECLASSPATH
```

Syntaxe :

```
java org.antlr.v4.Tool -visitor
              -o <output-folder>
                  <grammaire.g4>
```

Génération du code en Python sous Windows

- On crée un fichier batch qui appelle le générateur d'AntLR et permet de générer le code à partir d'un fichier grammaire g4
- antlr-generate.bat

```
@echo off
set SAVECLASSPATH=%CLASSPATH%

set CLASSPATH=lib/antlr-4.9.1-complete.jar

java -Xmx500M org.antlr.v4.Tool -visitor -o antlrPy
     -Dlanguage=Python3 grammar/Logo.g4
pause
set CLASSPATH=%SAVECLASSPATH%
```

Synchronisation avec Eclipse

- Synchroniser le code généré avec l'environnement de développement Eclipse.
  - Créer un projet Eclipse et un package pour les classes créées par le fichier batch ;
  - Mettre le fichier batch à la racine du projet
  - Créer la grammaire dans un répertoire (`grammar/Logo.g4`) ;
  - Utiliser l'option -o pour indiquer le répertoire où les fichiers sont générés.
  - Utiliser l'option -visitor pour générer aussi les classes pour le design pattern Visitor.

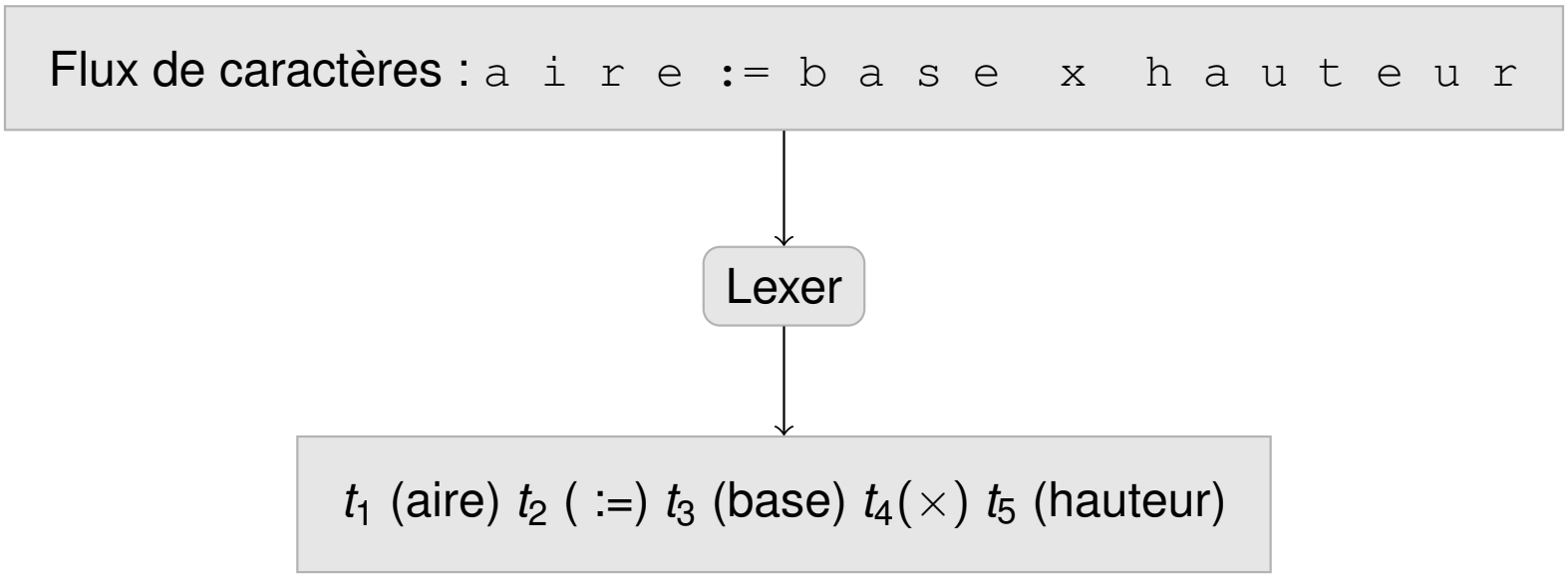
Sommaire

Analyses

- Analyse lexicale
  - Flux de caractères $\longrightarrow$ Flux de tokens
- Analyse syntaxique
  - Flux de tokens $\longrightarrow$ Interprétation
  - Flux de tokens $\longrightarrow$ Arbre de dérivation
- Traitement à partir de l'arbre de dérivation
  - Arbre de dérivation $\longrightarrow$ Interprétation

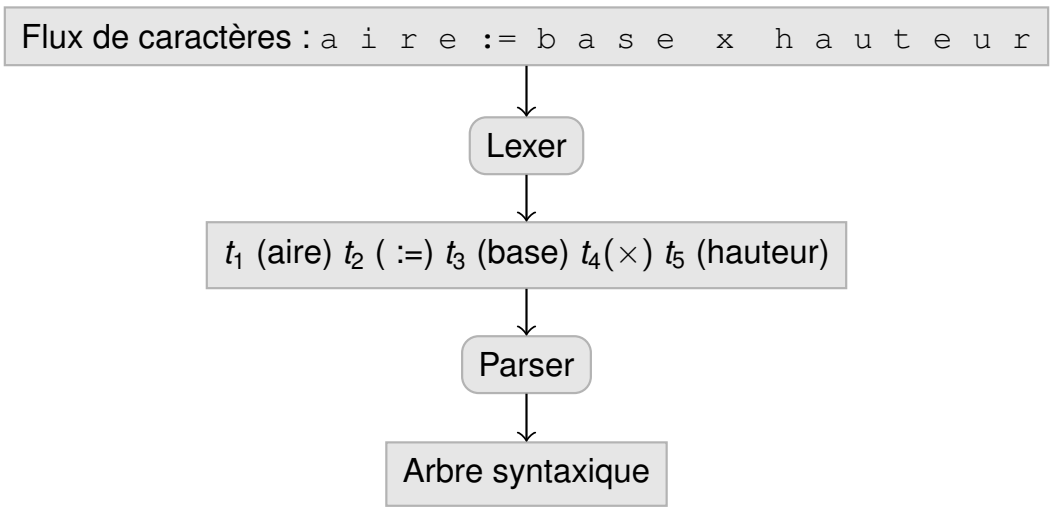
Analyse Lexicale

- L'analyseur lexical (lexer) est l'outil qui permet de découper un flux de caractères en un flux de mots du langage (tokens).

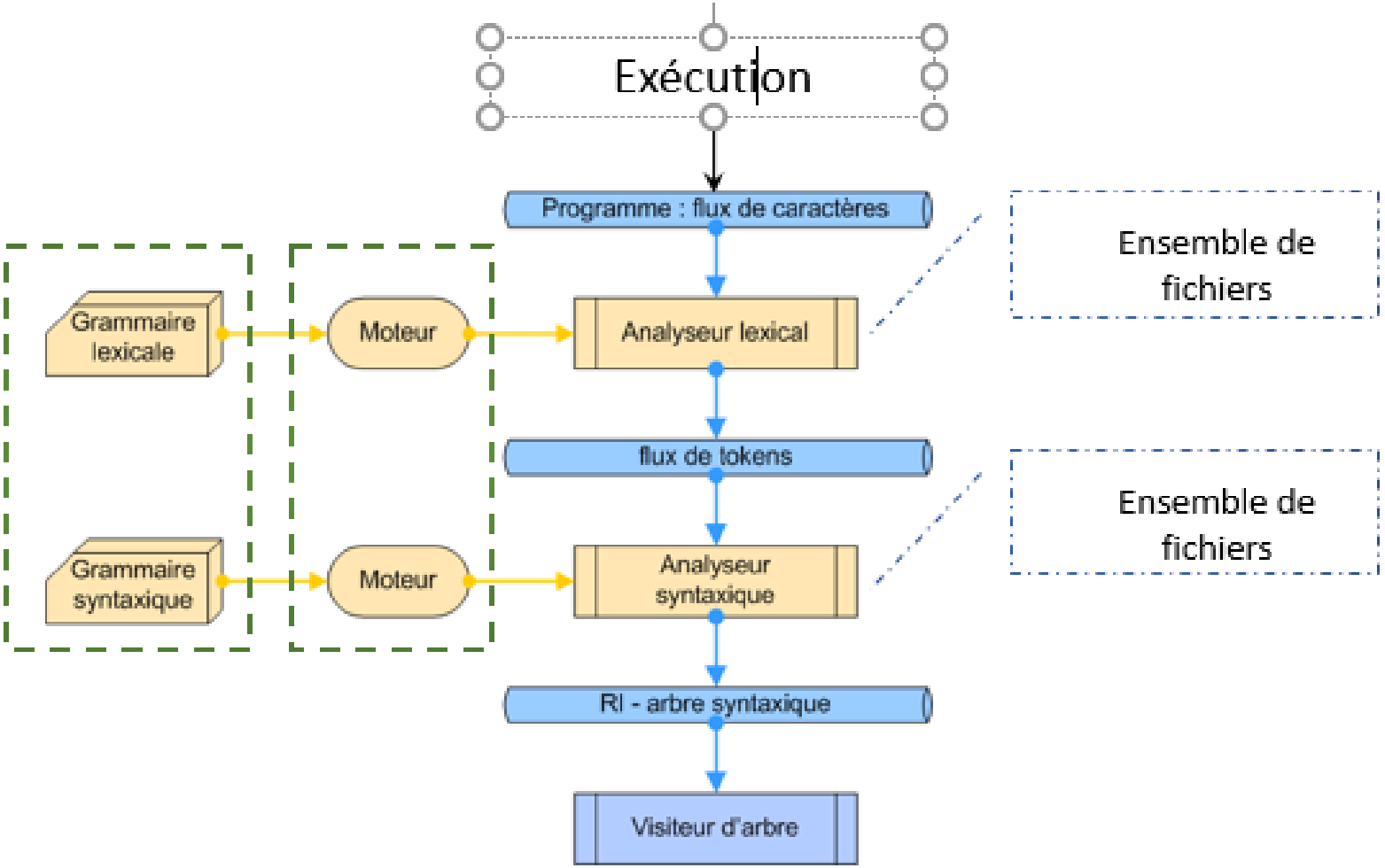


Analyse Syntaxique

- L'analyseur syntaxique (parser) vérifie que l'ensemble des mots issus de l'analyse lexicale (les tokens) forme une phrase syntaxiquement correcte. Il n'y a pas de garantie concernant la sémantique de cette phrase.



Génération


Exécution
Programme : flux de caractères
Grammaire lexicale
Moteur
Analyseur lexical
Ensemble de fichiers
flux de tokens
Grammaire syntaxique
Moteur
Analyseur syntaxique
Ensemble de fichiers
RI - arbre syntaxique
Visiteur d'arbre

Fichier grammaire : Logo.g4

```
grammar Logo;
@header { package logoparsing; }
FLOAT : [0-9][0-9]*('.'[0-9]+)? ;
WS : [ \t\r\n]+ -> skip ;

programme : liste_instructions ;
liste_instructions :
  (instruction)+
;
instruction :
   'av' expr      # av
 | 'td' expr      # td
;
expr :
   FLOAT          # float
 | '(' expr ')'   # parenthese
;
```

Sommaire

Règles

- Les catégories lexicale (lexer) sont en majuscule.
- Les catégories syntaxiques (variables) du parser sont en minuscule.
- `programme` est considéré comme l'axiome de la grammaire, mais rien ne l'indique.
- AntLr déclare automatiquement les tokens, mais il est possible de faire des catégories lexicales à un seul lexème.

Classes générées (exemple Java)

1. Lexer : `LogoLexer.java`
2. Parser : `LogoParser.java`
3. Listes de tokens : `Logo.tokens ; LogoLexer.tokens`
4. Listeners :
   `LogoListener.java` (interface)
   `LogoBaseListener.java` (classe implémentant à vide l'interface)
5. Visitors :
   `LogoVisitor.java` (interface)
   `LogoBaseVisitor.java` (classe implémentant à vide l'interface)

Parser

```
public class LogoParser extends Parser {
 public LogoParser(TokenStream input) {...}
 public final InstructionContext instruction()
     throws RecognitionException {...}
 public static class InstructionContext extends
      ParserRuleContext {...}
}
```

- Pour chaque règle syntaxique (ex : instruction) une méthode reprenant le nom de la règle et une classe contexte sont créées pour l'analyse d'une phrase.

Utilisation du parser

- Un parser ANTLR construit un arbre syntaxique à partir du programme donné en entrée.
- Il est nécessaire de parcourir cet arbre pour effectuer des actions.
- Traitements :
  - Simple : un seul parcours de l'arbre en profondeur d'abord est nécessaire pour le traitement. On utilise un listener d'événements.
  - Evolué : le parcours de l'arbre n'est pas nécessairement complet ; une sous-arborescence peut être parcourue plusieurs fois. On utilise un visiteur d'arbre.

Visitor

- ANTLR genère une interface visitor et une classe de base implémentant à vide l'interface.
  - Un visitor est un design pattern dont le but est d'exécuter une opération sur les éléments d'une structure (arbre) sans changer le type des éléments de la structure sur laquelle il opère.

```
public class LogoBaseVisitor<T> extends
  AbstractParseTreeVisitor<T> implements LogoVisitor<T>
  ...
  @Override
public T visitAv(LogoParser.AvContext ctx)
  { return visitChildren(ctx); }
  ...
}
```

Type du Visiteur

- ANTLR utilise un type paramètre pour la valeur de retour des méthodes du visiteur.
- Lors de la création de la classe visiteur il faut dériver la classe de base fournie par ANTLR avec le choix du type.
- Toutes les méthodes doivent retourner le même type de valeurs.

```
public class LogoTreeVisitor extends
    LogoBaseVisitor<Integer> {
    @Override
    public Integer visitAv(AvContext ctx) {
    ... // mettre le code
    }
...
}
```

Type de retour des méthodes

- Une règle expression a pour objectif de calculer un nombre (Double). Elle n'y réussit pas toujours (division par 0, logarithme de nombres négatifs, etc.).
- Les autres règles ont pour objectif de réaliser un processus (ex : règles des instructions).
- On ne peut utiliser le type de retour d'une méthode pour retourner une valeur calculée (ce ne serait possible que pour les règles expression et seulement si le calcul aboutissait).

Type de retour : Integer

- On utilisera le type Integer comme type de retour d'une méthode. Il indiquera un processus réussi (valeur 0) ou une erreur d'exécution (-1, -2, etc. selon le typage d'erreurs désiré).
- On utilisera une structure de type Map pour stoker les valeurs que certaines méthodes doivent calculer.

```
public class LogoTreeVisitor extends
    LogoBaseVisitor<Integer> {
...
}
```

Grammaire et méthode de type visit

```
liste_instructions :
    (instruction)+
;

public Integer visitListe_instructions(...) {
...
}
```

Grammaire et méthode de type visit

```
instruction :
    'av' expr # av
  | 'td' expr # td
;

public Integer visitAv(...) {...}
}
public Integer visitTd(...) {...}
}
```

Méthode statique minimale

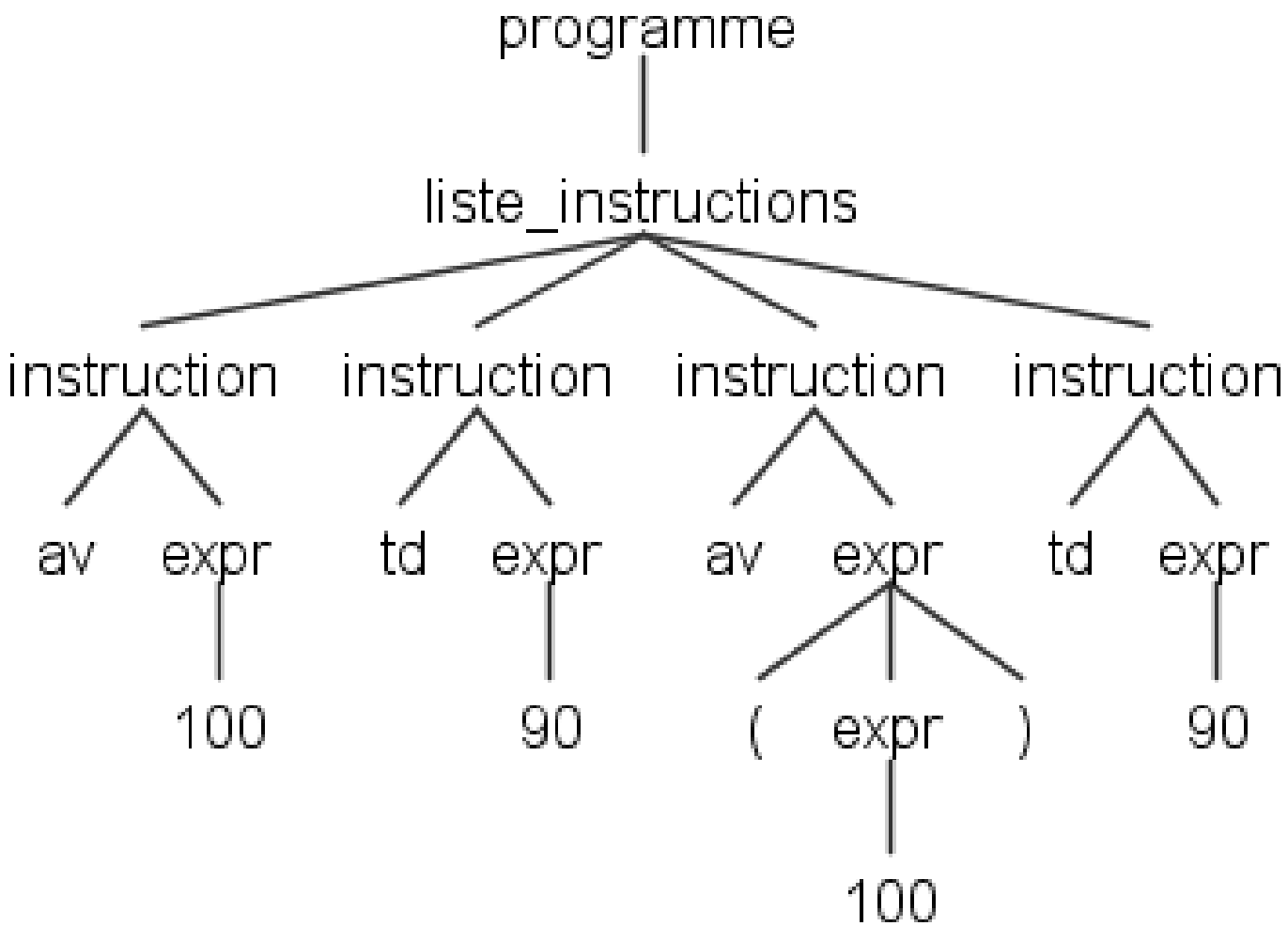
```
public static String program = "programs/logo-prg.txt";
public static void run_mini() {
 try {
  // lexer
  CharStream str = CharStreams.fromFileName(program);
  LogoLexer lexer = new LogoLexer(str);
  // parser
  CommonTokenStream tokens = new CommonTokenStream(lexer);
  LogoParser parser = new LogoParser(tokens);
  ParseTree tree = parser.programme();
  System.out.println(tree.toStringTree(parser));
 } catch(Exception ex) {
    ex.printStackTrace();
 }
}
```

Résultat

```
av 100 td 90 av (100) td 90
```

```
(programme
  (liste_instructions
    (instruction av (expr 100))
    (instruction td (expr 90))
    (instruction av (expr ( (expr 100) ) ) )
    (instruction td (expr 90))))
```

Génération


programme
liste_instructions
instruction
instruction
instruction
instruction
av
expr
td
expr
av
expr
td
expr
100
90
(
expr
)
90
100

Sommaire

Métalangage du lexer

- Opérateurs
  - concaténation
  - alternative : |
  - au moins 1 : +
  - 0 ou plus (Kleene) : *
  - 0 ou 1 : ?
  - négation : $\sim$
  - caractère quelconque : .
- Parenthèses pour forcer l'appel des opérateurs.

Facilités

- Fragment pour simplifier l'écriture de certaines règles :

```
fragment
LET : 'A'..'Z' | 'a'..'z' ;
fragment
DIGIT  :  '0'..'9' ;
ID : LET (LET | DIGIT)+ ;
SPACE : ' ' ;
```

- Pour ignorer les espaces, tabulations, retours à la ligne dans l'écriture d'un programme :

```
WS : [ \t\r\n]+ -> skip ;
```

Nommer chaque alternative

```
liste_instructions :
  (instruction)+
;
instruction :
    'av' expr # av
  | 'td' expr # td
;
```

- AntLR ne permet pas d'écrire plusieurs règles avec la même tête (variable), mais il offre une alternative de possibilités pour exprimer une variable.
- Il est nécessaire de donner un nom à chaque cas d'une alternative pour générer une méthode adaptée du visiteur. On aura l'instruction # av, l'instruction # td, etc.

Accès aux nœuds de l'arbre

- Un parser ANTLR construit un arbre syntaxique à partir du programme donné en entrée.
- Les méthodes des listener et visitor créés par ANTLR ont un paramètre (ctx) dont le type est une classe décrivant le nœud de l'arbre créé par le parser.
- Ce paramètre permet d'accéder aux nœuds fils.
  - Depuis `visitProgramme` on a accès à `ctx.liste_instructions()`.
  - Plusieurs fils de même nom sont accessibles dans une liste : depuis `visitListe_instructions` on a accès à `ctx.instruction()`.
  - Accès aux tokens par leur nom : depuis `visitFloat` on a accès à `ctx.FLOAT()`.

Forme générale d'une grammaire

- Déclaration de la grammaire
  - grammar <GrammarName> ;
- Section @header{...}
  - déclaration de package

```
@header {
  package logoparsing;
}
```

- Règles lexicales (Majuscule)
- Règles syntaxiques (Minuscule)

Fichier grammaire : Logo.g4

```
// lexical
FLOAT : [0-9][0-9]*('.'[0-9]+)? ;
WS : [ \t\r\n]+ -> skip ;
// syntaxique
programme :
 liste_instructions
;
liste_instructions :
 (instruction)+
;
instruction :
   'av' expr # av
 | 'td' expr # td
;
expr :
   FLOAT          # float
 | '(' expr ')'   # parenthese
;
```

Principe du visitor

- L'arbre de dérivation est entièrement construit lorsque le visiteur est lancé.
- Le paramètre contexte (ctx) de chaque méthode donne accès aux éléments fils du nœud courant.
- Le visitor définit la méthode de visite de chaque nœud de l'arbre (règle de la grammaire = av, méthode = visitAv).
- Une méthode peut :
  - appeler la méthode de visite des fils du nœud courant (visitChildren(ctx)).
  - appeler la méthode de visite sur les nœuds fils pertinents (visit(ctx.expr()).
  - ou appeler la visite d'un nœud mémorisé précédemment.

Valeur calculée

- Evaluer une expression a pour but de produire un résultat de type Double s'il n'y a pas d'erreurs, de vérifier les erreurs d'exécution possibles et de stocker la valeur de l'expression.
- La méthode de visite d'une expression retourne une valeur entière qui n'est pas le résultat de l'expression.
- La valeur retournée indique une erreur potentielle (0 = ok, n < 0 = type d'erreur)

Visite d'une instruction

- Grammaire.

```
    'av' expr   # av
  | 'td' expr   # td
```

- Il suffit de visiter le nœud expr.

```
public Integer visitAv(AvContext ctx) {
 int n = visit(ctx.expr());
 // récupération de l'erreur suivant n,
 // récupération de la valeur calculée si n == 0
 return 0; // récupération de l'erreur
}
```

Évaluation d'une expression parenthèse

- Il suffit de visiter le nœud expression fils.

```
public Double visitParenthese(ParentheseContext ctx) {
 int n = visit(ctx.expr());
 if (n == 0)
   ...; // récupération de la valeur calculée
        // sauvegarde de la valeur de la parenthèse
 return n;
}
```

Programme

```
public static void run_default_visitor() {
 try {
  CharStream stream =
          CharStreams.fromFileName("programs/logo-prg.txt");
  LogoLexer lexer = new LogoLexer(stream);
  CommonTokenStream tokens = new CommonTokenStream(lexer);
  LogoParser parser = new LogoParser(tokens);
  ParseTree tree = parser.programme();
  LogoTreeVisitor visitor = new LogoTreeVisitor();
  visitor.visit(tree);
 } catch(Exception ex) {
   ex.printStackTrace();
 }
}
```

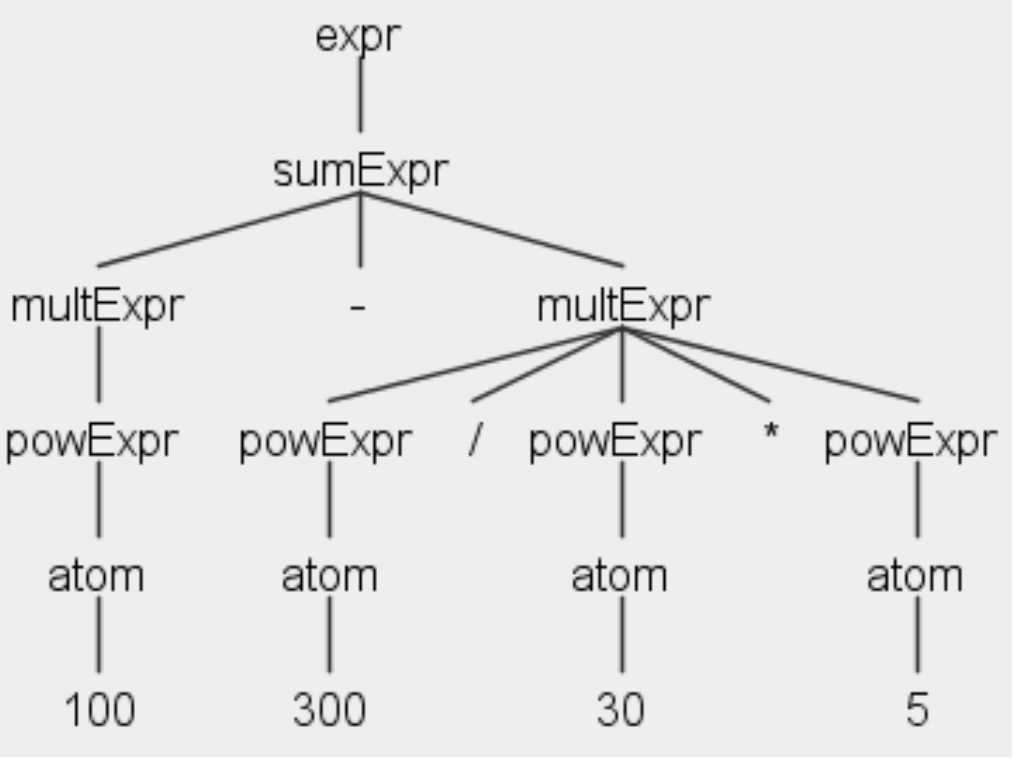
Conditions

- Dans les langages de programmation, la grammaire doit permettre d'écrire et d'évaluer correctement des expressions aritmétiques :
  - fixer la priorité des opérateurs
  - assurer l'associativité à gauche
  - utiliser des parenthèses pour forcer certains calculs
- 100 - 30 + 20 doit être évalué à 90
- 100 - 30 * 2 doit être évalué à 40
- 100 - 300 / 30 * 5 doit être évalué à 50

Grammaire LL

```
expr : sumExpr ;
sumExpr :  multExpr (('+' | '-') multExpr)*
    ;
multExpr : powExpr (('*' | '/') powExpr)*
    ;
powExpr : atom ('^' atom)*
;
atom :
    FLOAT          #  float
 |  '(' expr ')'   # parenthese
;
```

Exemple : 100 - 300 / 30 * 5



Analyse descendante

- Dans le cas de grammaire de type LL, les règles ne peuvent être récursives à gauche. Ce qui complique leur écriture.
- Les règles des opérateurs les moins prioritaires doivent être appelées avant lors du parsing.
- Les opérateurs de même force doivent se retrouver dans les mêmes règles.
- Plus de deux opérandes fils peuvent apparaître dans l'arbre alors que chaque opérateur binaire n'a que deux fils.

Evaluation simplifiée d'une expression produit

```
public Integer visitMultExpr(MultExprContext ctx) {
 int operandRow = 0;
 int nb = ctx.powExpr().size();

 Double result = evaluer(ctx.powExpr(l));
 operandRow++;
 int operatorRow = 1;
 while (operandRow < nb) {
  Double right = evaluer(ctx.powExpr(l));
  result =  ctx.getChild(operatorRo)
            .getText().equals("*") ?
     result * right : result / right;
  operandRow++; operatorRow +=2 ;
 }
 // sauver(ctx, result);
 return 0;
}
```

Sommaire

Méthode

- On utilise une structure de type Map où l'on mettra la valeur calculée pour chaque nœud expression (clé de la map).
- ANTLR fournit une classe `ParseTreeProperty` structure des map d'objets indexés par les nœuds de l'arbre.
- Le visiteur définit la structure.
- Chaque méthode stocke dans la map la valeur qu'elle calcule.
- Chaque méthode va chercher dans la map les valeurs dont elle a besoin.
- Chaque méthode retourne un entier qui indique si le calcul s'est bien passé.

Valeur calculée

```
public class LogoTreeVisitor extends
    LogoBaseVisitor<Integer> {
    ParseTreeProperty<Double> atts =
                    new ParseTreeProperty<>();
// Accesseurs sur cette structure
 public void setValue(ParseTree node, double value) {
  atts.put(node, value);
 }
 ...
}
```

Accesseur en lecture

Il est possible de lever une exception lors de la phase de mise au point, lorsque un nœud n'est pas trouvé dans la map (et qu'il devrait y être).

```
public Double getExprValue(ParseTree node) {
 Double value = atts.get(node);
 if (value == null) {
  throw new NullPointerException();
 }
 return value;
}
```

Sommaire

Avancées de ANTLR v4

- ANTLR v4 simplifie l'écriture de règles de grammaire grâce à l'algorithme ALL(*)
- Il élimine les conflits principaux des grammaires LL et donc des parser descendants :
  - Non factorisation à gauche

```
a :
    b INT
  | b '(' a ')'
;
```

  - Récursivité directe à gauche

```
expr : expr * expr;
```

Grammaire v4

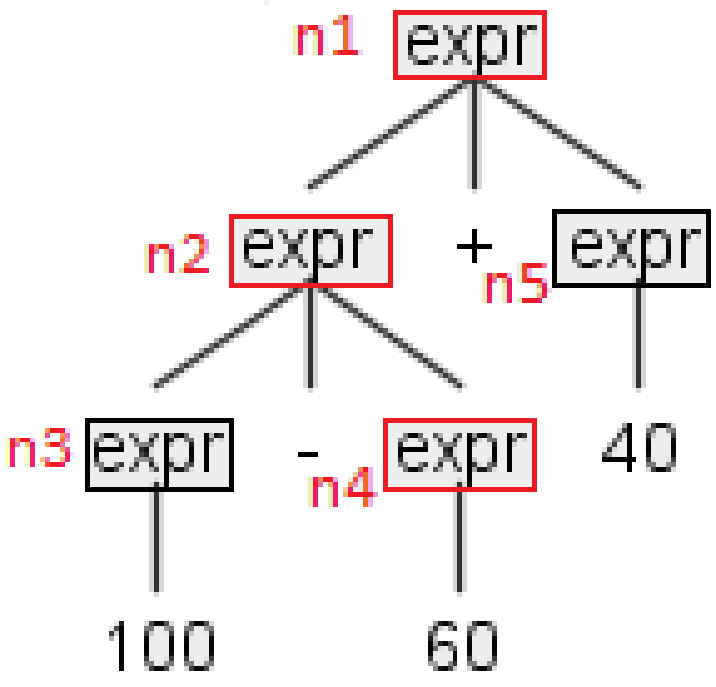
```
grammar Logo;
@header {
  package logoparsing;
}
FLOAT : [0-9][0-9]*('.'[0-9]+)? ;
...
instruction :
   'av' expr # av
 | 'td' expr # td
;
expr:
      expr ('*' | '/' ) expr    #mult
   |  expr ('+' | '-' ) expr    #sum
   |  FLOAT                     #float
   |  '(' expr ')'              #parenthese
;
```

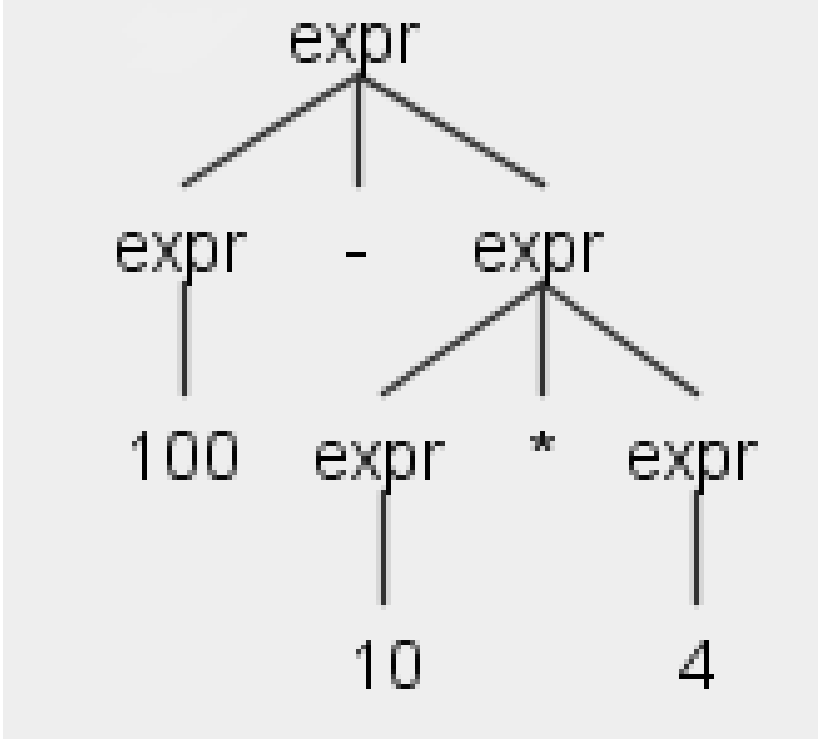
Interprétation

```
expr:
      expr ('*' | '/' ) expr   #mult
    | expr ('+' | '-' ) expr   #sum
    ...
    ;
```

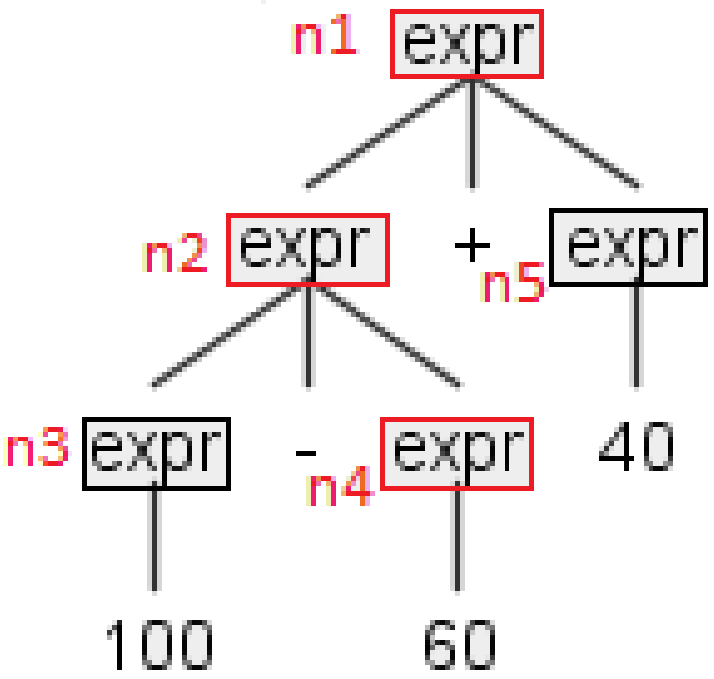
- La récursivité directe ne pose pas de problème.
- Le premier cas de l'alternative est considéré prioritaire ; il faut y mettre l'opérateur considéré le plus important (*, /).
- L'associativité à gauche est respectée.
- ANTLR construit l'arbre qui permet une juste interprétation à partir de ces méta-règles.
- 100 - 60 + 40 = 80 ; 100 - 10 * 4 = 60

Exemples : 100 - 60 + 40 = 80 ; 100 - 10 * 4 = 60





Exemple : 100 - 60 + 40 = 80



n3	100	float
n4	60	float
n2	40	sum
n5	40	float
n1	80	sum

Evaluation d'un nombre

- Elle correspond à l'alternative float.

```
public Integer visitFloat(FloatContext ctx) {
 String fText = ctx.FLOAT().getText();
 setExprValue(ctx, Float.parseFloat(fText));
 return 0;
}
```

- Il faut sauvegarder la valeur dans la map.

Autres cas

- Dans les autres cas d'expressions et dans les instructions qui nécessitent une expression, il faut synthétiser les résultats obtenus par le (les) nœud(s) expression fils :
  - visiter le (les) nœud(s) expression fils. Les visites récursives feront remonter les erreurs éventuelles grâce aux valeur de retour des méthodes.
  - aller chercher la valeur de l'expression dans la map si le calcul est valide (valeur de retour de la visite)
- Il faut deux informations : validité du calcul et le cas échéant valeur de l'expression.

Couple d'évaluation

```
private Pair<Integer, Double>
                    evaluate(ParseTree expr) {
 int b = visit(expr);
 Double val =    b == 0 ? getValue(expr) :
                          Double.POSITIVE_INFINITY;
 return new Pair<Integer, Double>(b, val);
}
```

Evaluation d'une parenthèse

- Elle correspond à l'alternative parenthese.

```
public Integer visitParenthese(ParentheseContext ctx) {
 Pair<Integer, Double> exp = evaluate(ctx.expr());
 if (exp.a == 0) {
  setValue(ctx, exp.b);
 }
 return exp.a;
}
```

Evaluation d'une expression somme

- Liste des 2 nœuds fils opérandes : `ctx.expr()`.
- Il se peut que l'un des deux termes n'a pas pu être évalué (cas de la division par 0 : 100 + 10 / 0).
- Il n'est pas possible d'aller chercher la valeur du terme dans la map.

Evaluation d'une expression somme

```
public Integer visitSum(SumContext ctx) {
Pair<Integer, Double> left, right;
  left = evaluate(ctx.expr(0));
  right = evaluate(ctx.expr(1));
  if (left.a == 0 && right.a == 0) {
    String sign = ctx.getChild(1).getText();
    Double r = sign.equals("+") ? left.b + right.b :
                                  left.b - right.b;
    setValue(ctx, r);
    return 0;
  }
  return left.a == 0 ? right.a : left.a;
}
```

Evaluation d'une expression produit

- L'évaluation d'un produit est basée sur le même principe mais il faut considérer en plus le cas de la division par 0.

Synthèse et héritage

Lors de la visite d'un arbre en profondeur d'abord, les méthodes peuvent avoir les caractéristiques :

- Synthèse : la méthode visite les noeuds fils et produit un résultat synthèse des résultats produits lors des visites des nœuds fils.
- Héritage : la méthode a besoin de valeurs héritées de la visite des n œuds ancêtres ou frères (siblings) précédents.
- Le visiteur doit définir des structures utilisées lors des visites des nœuds : maps, pile, etc.