Théorie des Langages

AntLR - Générateur d'analyseurs

Claude Moulin

Université de Technologie de Compiègne

Printemps 2012

Sommaire

AntLR

- ANTLR : ANother Tool for Language Recognition
- URL : http ://www.antlr.org
- Auteur : Terence Parr (MageLang Institute)
- Mis au point par Terence Parr en 1994-1995, après sa thèse au sein de l'universite Purdue, Indiana, USA.
- Constitue en fait une extension du système intitulé PCCTS (Purdue Compiler Construction Tool Set).
- Dans l'idée de ses concepteurs, il devait, en plus de reconnaître des langages, permettre de disposer d'un parser qui interagisse avec le lexer, fournir un rapport d'erreur assez explicite et construire 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 (appelé fichier grammaire) contenant les règles définissant le langage.
  - Code généré : classes en C++, Java ou C#.
  - Le fichier grammaire contient également des actions : instructions insérées dans le code généré et qui doivent être compatibles avec le langage cible.
- AntLR est aussi un Méta Langage généralisé utilisé pour écrire le fichier grammaire.

Sommaire

Ligne de commande

- ANTLR est une application JAVA ;
- ANTLR prend en entrée un fichier grammaire (extension .g) et produit en sortie un ensemble de fichiers (dans le langage cible choisi) ;
- Ligne de commande :
  - java antlr.Tool grammaire.g ;
  - Java version 1.5 ou supérieure ;
  - requiert antlr-3.3-complete.jar

ANTLRWorks

- ANTLRWorks est un outil graphique de développement et de deboggage des grammaires AntLR version 3.
- ANTLRWorks permet de :
  - Écrire une grammaire ;
  - Construire l'arbre de dérivation ;
  - Tester les règles de la grammaire sur des exemples ;
  - Générer le code du parser.

Plug-in Eclipse

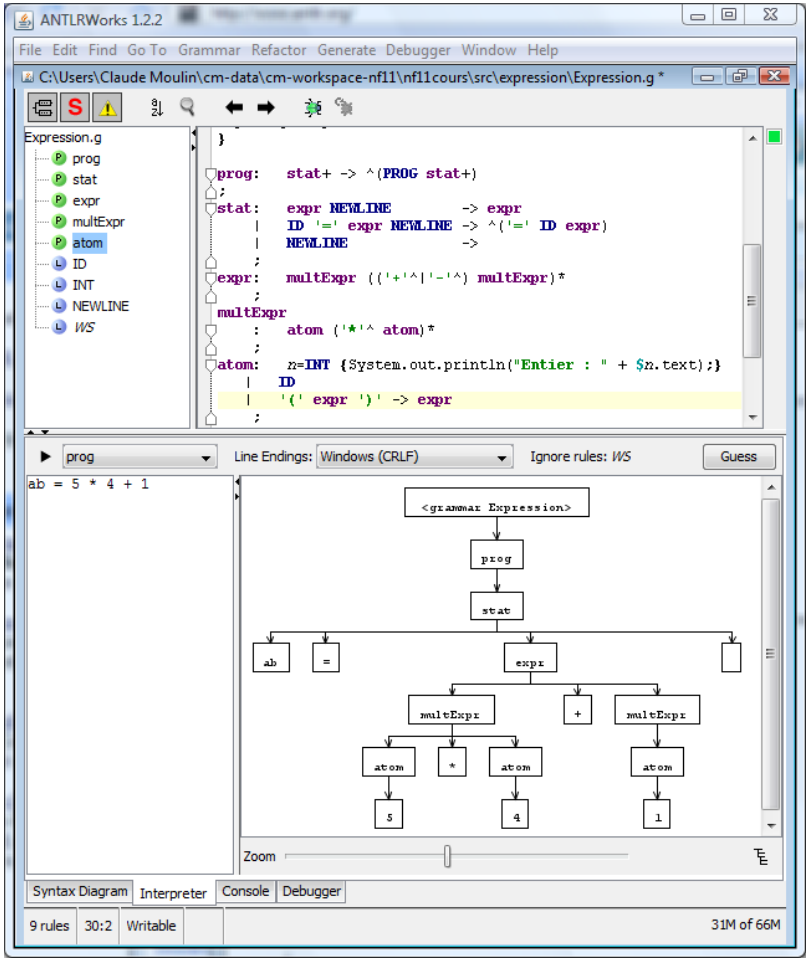
- ANTLR IDE
- L'adresse `http://goo.gl/YbZV5` renvoie à : `http://antlrv3ide.sourceforge.net/`
- Plug-in eclipse servant à éditer des grammaires ANTLR version 3 et à générer le code des parsers correspondant.
- `http://antlrv3ide.sourceforge.net/updates/3.6`

Installation de ANTLRWOKS

- Télécharger la librairie ANTLRWORKS : antlrworks-1.4.2.jar
- Installer dans un répertoire : lib
- Créer un fichier batch :

```
@echo off
set SAVECLASSPATH=%CLASSPATH%
set CLASSPATH=lib/antlrworks-1.4.2.jar;
set javapath =
      C:/"Program Files"/Java/jre1.5.0_11/bin
%javapath%/java org.antlr.works.IDE
set CLASSPATH=%SAVECLASSPATH%
```

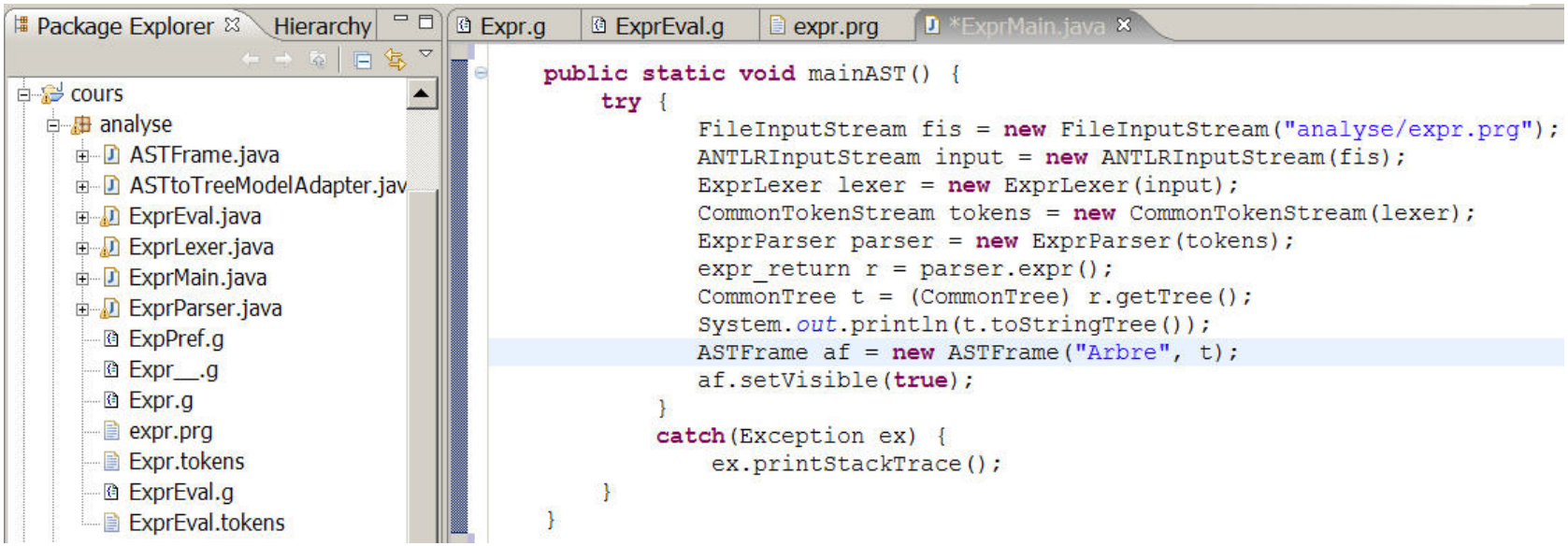
AntLRWorks


ANTLRWorks 1.2.2
File Edit Find Go To Grammar Refactor Generate Debugger Window Help
C:\Users\Claude Moulin\cm-data\cm-workspace-nf11\nf11cours\src\expression\Expression.g *
Expression.g
prog
stat
expr
multExpr
atom
ID
INT
NEWLINE
WS
prog: stat+ -> ^(PROG stat+)
stat: expr NEWLINE -> expr
ID '=' expr NEWLINE -> ^('=' ID expr)
NEWLINE ->
expr: multExpr (('+'^|'-'^) multExpr)*
multExpr
atom ('*'^ atom)*
atom: n=INT {System.out.println("Entier : " + $n.text);}
ID
'(' expr ')' -> expr
prog
Line Endings: Windows (CRLF)
Ignore rules: WS
Guess
ab = 5 * 4 + 1
<grammar Expression>
prog
stat
ab
=
expr
multExpr
+
multExpr
atom
*
atom
atom
5
4
1
Zoom
Syntax Diagram
Interpreter
Console
Debugger
9 rules
30:2
Writable
31M of 66M

Synchronisation avec Eclipse

- Synchroniser la sortie de ANTLRWORKS avec l'environnement de développement Eclipse.
  - Créer un projet Eclipse et un package pour les classes créées par ANTLRWORKS ;
  - Créer la grammaire dans ce répertoire (<Nom_Grammaire.g>) ;
  - Configurer ANTLRWORKS (File/Preferences/Output path) si nécessaire ;
  - Les fichiers Classe créés par ANTLRWORKS sont visibles dans Eclipse :
    - Vérification de la syntaxe des actions de la grammaire ;
    - Développement d'une application utilisant le parser.

Eclipse



- Installer dans le projet la librairies suivante. On peut créer un répertoire visible du projet en cours, copier les librairies dans ce répertoire et les ajouter au projet (Build Path/Add to Build Path).
  - antlr-3.3-complete.jar

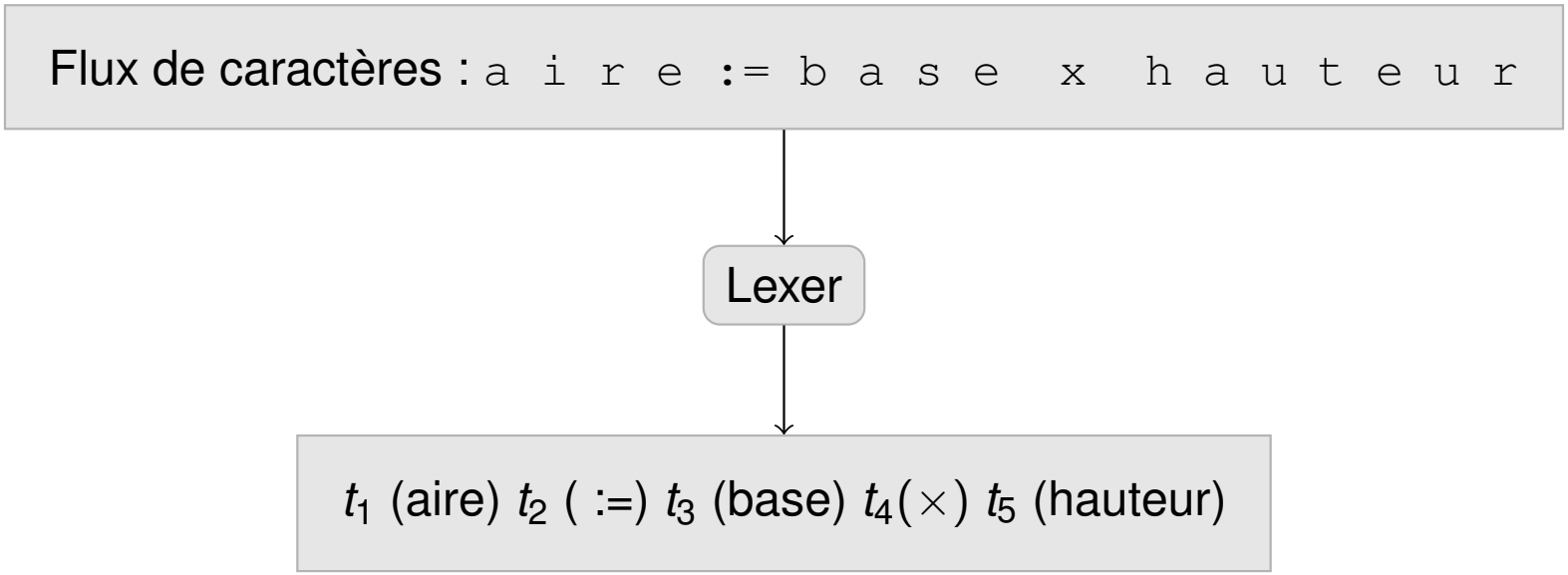
Sommaire

3 Analyses

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

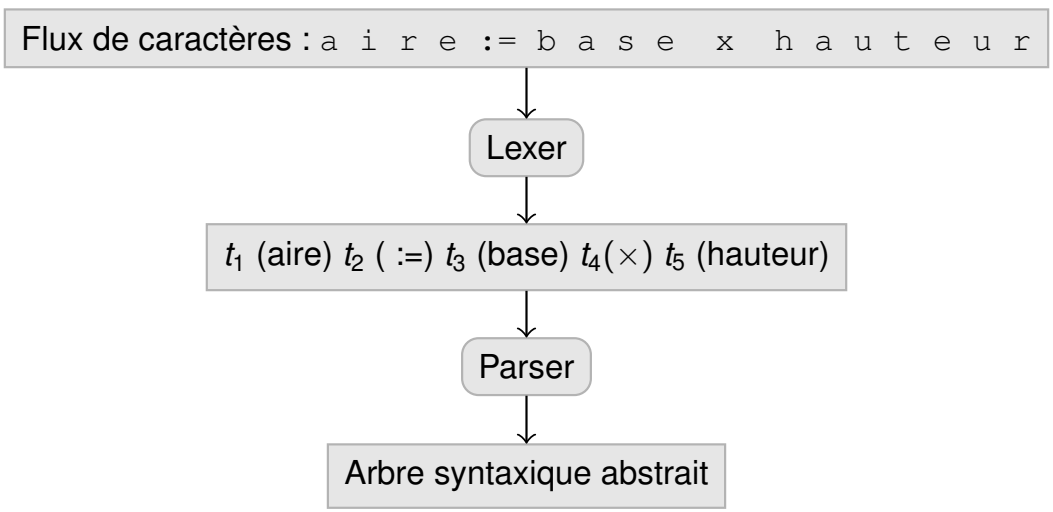
Analyse Lexicale

- 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 (tokens) forme une phrase syntaxiquement correcte. Il n'y a pas de garantie concernant la sémantique de cette phrase.



Exemple

- Grammaire permettant d'écrire des expressions arithmétiques du type :
  - 10 - 3 * 5
  - (20 - 10) / 4
- Symboles opératoires : $+$, $-$, $\star$, $/$, $\hat{}$;
- Les opérandes sont des nombres entiers ;
- Il est possible d'utiliser des parenthèses ;
- Les espaces et tabulations sont ignorés mais servent de séparateurs.

Fichier grammaire : Expr.g

```
grammar Expr;
tokens{
  PLUS = '+';
  MOINS = '-';
}
@lexer::header {
  package analyse;
}
@header {
  package analyse;
}
INT :   '0'..'9'+ ;
WS  :   (' '|'\t')+ {skip();} ;
expr : sumExpr ;
...
```

Sommaire

Règles

- Les catégories lexicale (lexer) sont en majuscule.
- Les catégories syntaxiques (variables) du parser sont en minuscule.
- `expr` est considéré comme l'axiome de la grammaire.
- Pour plus de lisibilité, les symboles peuvent être déclarés en tant que catégories lexicales dans la section `tokens`.
- AntLr déclare automatiquement les tokens.

Classes générées

1. Lexer

```
package analyse;
public class ExprLexer extends Lexer
  public ExprLexer(CharStream input) {...}
  ...
  public final void mINT() {...}
}
```

2. Parser

```
public class ExprParser extends Parser {
 public ExprParser(TokenStream input) {...}
 public final void expr()... {...}
}
```

Programme minimal

Méthode appelée par la méthode main d'une classe Java :

```
public static void mainvoid() {
 try {
  FileInputStream fis =
      new FileInputStream("analyse/expr.prg");
  ANTLRInputStream input = new ANTLRInputStream(fis);
  ExprLexer lexer = new ExprLexer(input);
  CommonTokenStream tokens =
                          new CommonTokenStream(lexer);
  ExprParser parser = new ExprParser(tokens);
  parser.expr();
 }
 catch(Exception ex) {
  ex.printStackTrace();
 }
}
```

Sommaire

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(); } ;
```

Parser Expr

```
expr : sumExpr
     ;
sumExpr:    multExpr ((PLUS|MOINS) multExpr)*
     ;
multExpr
     :    powExpr (('*'|'/') powExpr)*
     ;
powExpr
     :    atom ('^' atom)*
     ;
atom:     INT
      |   '(' expr ')'
     ;
```

Actions

- Une action se traduit par du code inséré dans les méthodes des classes générées par AntLR.

```
WS  : (' '|'\t'|('\r'? '\n'))+
      { skip(); } ;
atom: n=INT
    {System.out.println("n=" + $n.getText());}
    | '(' expr ')'
;
```

- Les tokens (ex : INT) peuvent être récupérés dans des variables (ex : `n`). La méthode principale est `getText()` qui retourne le lexème attaché au token (utiliser `$n` dans le code de l'action).

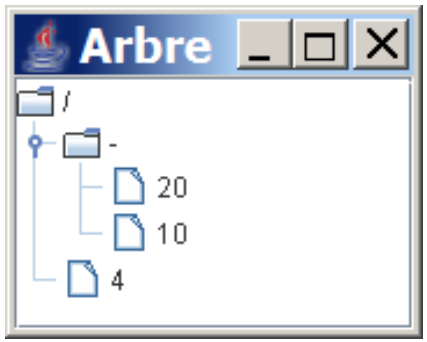
Sommaire

Arbre abstrait : objectifs

- Garder dans une structure les tokens les plus significatifs et seulement ceux-là.
- Encoder dans une structure d'arbre (2 dimensions), la structure grammaticale utilisée par le parser pour matcher les tokens associés (les noms des règles sont superflus).
- Avoir une structure telle qu'il soit simple pour un programme de la reconnaître et de naviguer à l'intérieur.

Arbre abstrait

- Programme : (20 - 10) / 4
- Arbre abstrait : (/ (- 20 10) 4)



- Interprétation : 2.5

Construction de l'AST par AntLR

```
grammar Expr;
options {
  output = AST;
}
expr : sumExpr ;
sumExpr:   multExpr ((PLUS^|MOINS^) multExpr)*
    ;
multExpr
    :   powExpr ((MULT^|DIV^) powExpr)*
    ;
powExpr
    :   atom (EXP^ atom)*
    ;
atom:   INT
    |   '('! expr ')'!
    ;
```

Construction automatique - 1

- Ajouter l'option de construction automatique de l'arbre.

```
grammar Expr;
options {
    output=AST;
}
```

- Durant le parsing, chaque règle permet de créer une nouvelle arborescence qui vient se greffer sur le noeud courant.
- Par défaut chaque token sert à créer un noeud de l'arbre.
- Les feuilles sont automatiquement insérées dans l'arbre.

Construction automatique - 2

- PLUS^ : le symbole devient le nœud racine d'une nouvelle arborescence.
- '(' ! : le symbole (n'est pas inséré dans l'arbre.
- A B^ C donne l'arbre ^(B A C), arbre de racine B ayant pour fils A et C
- A B^ C^ donne l'arbre ^(C ^(B A)).

Construction automatique - 3

Lorsque l'option `output=AST` est présente dans la grammaire, pour chaque règle de la grammaire, AntLR crée :

- une méthode,
- une classe interne contenant l'arbre AST construit et visible au niveau de la règle
- et chaque méthode retourne une instance de cette classe

Pour la règle `expr`

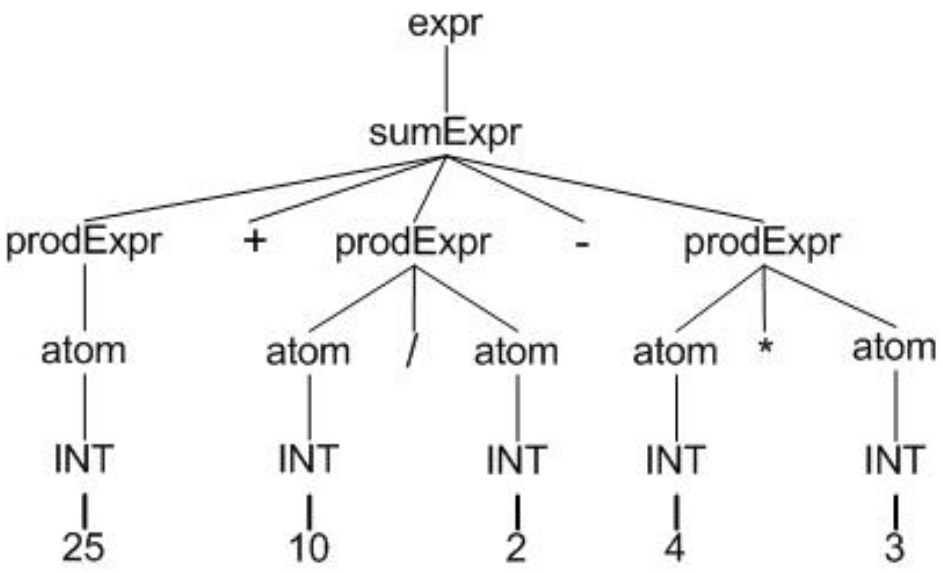
- `expr_return` est la classe
- `expr_return.getTree()` est la méthode retournant l'arbre
- `(CommonTree)` est le type de l'objet arbre retourné
- la méthode `expr` a pour signature
  `public final expr_return expr()`

Programme

```
public static void mainAST() {
 try {
  FileInputStream fis =
     new FileInputStream("analyse/expr.prg");
  ANTLRInputStream input =
            new ANTLRInputStream(fis);
  ExprLexer lexer = new ExprLexer(input);
  CommonTokenStream tokens =
               new CommonTokenStream(lexer);
  ExprParser parser = new ExprParser(tokens);
  expr_return r = parser.expr();
  CommonTree t = (CommonTree) r.getTree();
  System.out.println(t.toStringTree());
  ASTFrame af = new ASTFrame("Arbre", t);
  af.setVisible(true);
 }
 catch(Exception ex) {
  ex.printStackTrace();
 }
}
```

Construction de l'AST - 1

- Expression : 25 + 10 / 2 - 4 * 3

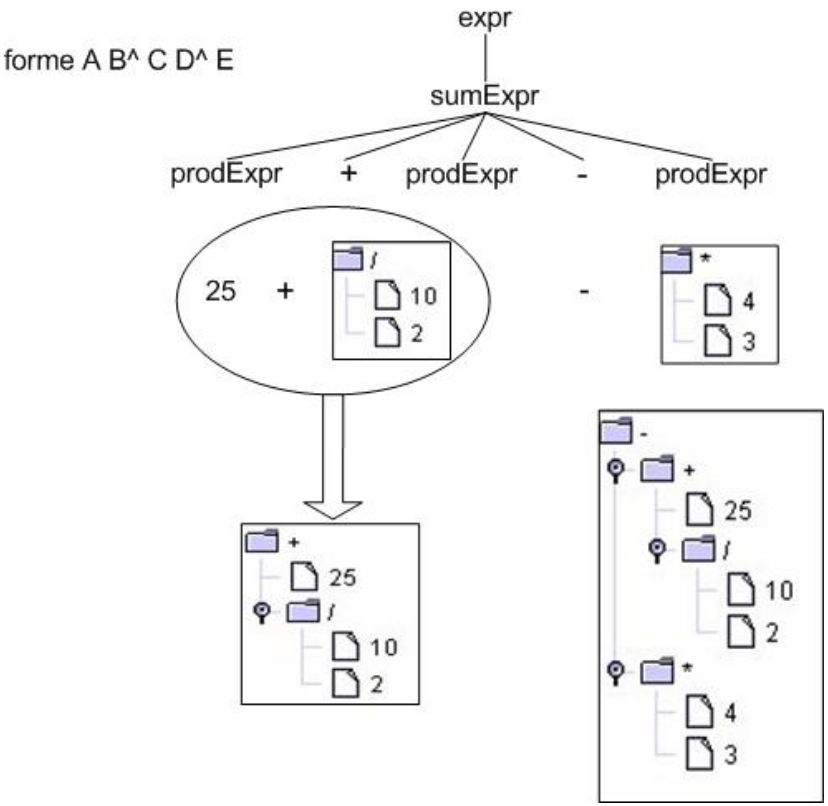


forme A B^ C


```
multExpr
    : powExpr
      ((MULT^|DIV^)
      powExpr)*
    ;
```

Construction de l'AST - 2

- Expression : 25 + 10 / 2 - 4 * 3



Construction manuelle

- On peut utiliser la construction automatique en utilisant les symboles `^` et `!` dans certaines règles ou certaines alternatives.
- On peut indiquer l'arborescence à construire :

```
atom:    INT
     |   '(' expr ')' -> expr
     ;
```

Si l'atome est un entier il est inséré sinon seul l'arbre issu de l'expression expr est inséré, sans les parenthèses.

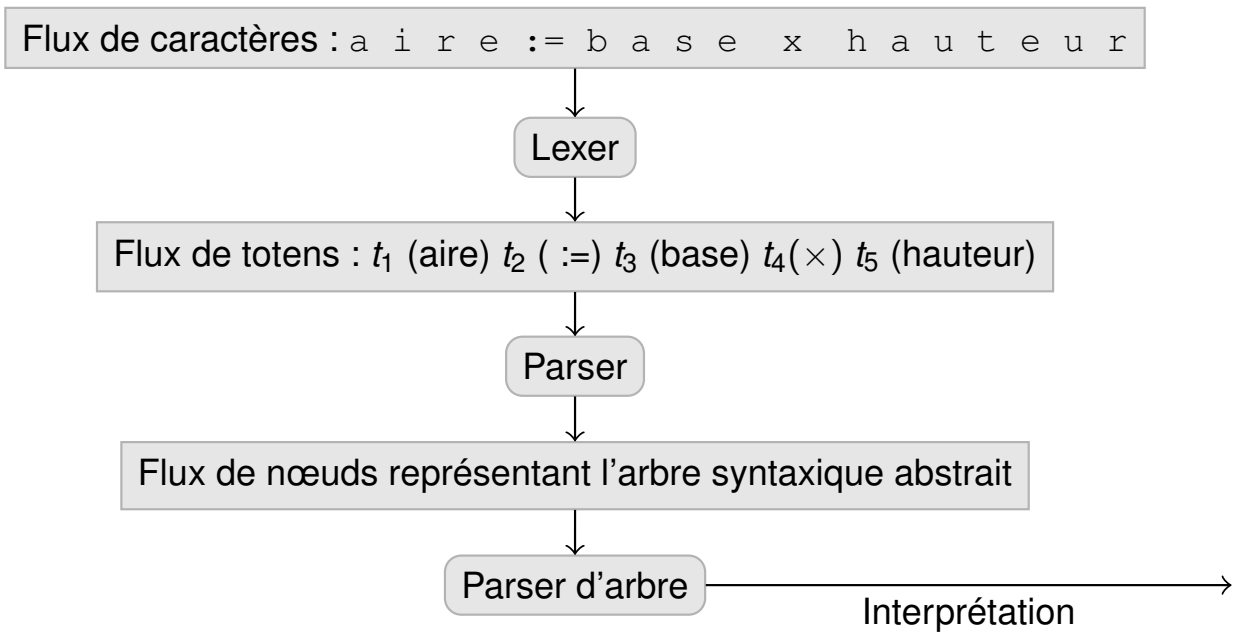
- Il est possible d'ajouter des noeuds dans l'arbre à partir de tokens imaginaires, déclarés dans la section `tokens` :

```
expr : sumExpr -> ^(SOMME sumExpr)
```

Sommaire

Sommaire

Analyse Sémantique



Forme de l'arbre

- On a construit un arbre où chaque noeud racine est associé à un token représentant un opérateur binaire : `+`, `-`, `*`, `/`, `^` et chaque feuille est un entier.
- Il faut définir pour chaque cas la valeur à retourner :
  - cas d'un entier : la valeur en flottant de l'écriture de l'entier ;
  - cas d'un opérateur : la valeur de l'opération à partir des valeurs des opérandes ;
  - la valeur des opérandes est donnée par l'appel à la même fonction.
  - la méthode doit donc retourner un nombre flottant.

Interprétation

- Arbre AST : (/ (- 20 10) 4)
- Règle principale :

```
exp returns [double v] :
    ^(PLUS x=exp y=exp)   {$v = $x.v + $y.v;}
  | ^(MOINS x=exp y=exp)  {$v = $x.v - $y.v;}
  | ^(MULT x=exp y=exp)   {$v = $x.v * $y.v;}
  | ^(DIV x=exp y=exp)    {$v = $x.v / $y.v;}
  | ^(EXP x=exp y=exp)
          {$v = Math.exp($y.v * Math.log($x.v));}
  | INT {$v = Double.parseDouble($INT.text);}
;
```

Grammaire d'arbre

- La grammaire

```
tree grammar ExprEval;
options {
 tokenVocab = Expr;
 ASTLabelType=CommonTree;
}
@header {
 package analyse;
}
exp returns [double v] :
...
   | INT {$v = Double.parseDouble($INT.text);}
;
```

- génère un parser :

```
public class ExprEval extends TreeParser
 public ExprEval(TreeNodeStream input) {...}
 public final double exp() ... {...}
}
```

Programme

```
public static void mainEval() {
  try {
   FileInputStream fis =
      new FileInputStream("analyse/expr.prg");
   ANTLRInputStream input = new ANTLRInputStream(fis);
   ExprLexer lexer = new ExprLexer(input);
   CommonTokenStream tokens =
      new CommonTokenStream(lexer);
   ExprParser parser = new ExprParser(tokens);
   expr_return r = parser.expr();
   CommonTree t = (CommonTree) r.getTree();
   CommonTreeNodeStream nodes =
      new CommonTreeNodeStream(t);
   ExprEval treeparser = new ExprEval(nodes);
   System.out.println(treeparser.exp());
  }
  catch(Exception ex) {
   ex.printStackTrace();
  }
}
```

Sommaire

Forme des règles

```
rulename[type id] returns [type id]
  @init  {init-action}
  @after {action-finale}
   :
     {action-debut}
      alt1 {action}
  |  {action} alt2 {action}
  |  {action} alt3 {action}
  |  ...
;
```

Forme générale d'une grammaire - 1

- Déclaration de la classe
  - (tree) grammar GrammarName ;
- Section options{...}
  - output = AST ; pour une construction de l'AST
  - tokenVocab=LexerName ; pour indiquer le nom du lexer
  - ASTLabelType=CommonTree ; pour indiquer le type des noeuds de l'arbre
- Section @header{...}
  - importation des classes nécessaires
  - déclaration de package
  - @lexer : :header pour le header du lexer

Forme générale d'une grammaire - 2

- Section tokens{...}
  - permet d'ajouter des tokens imaginaires ou non.
- Section @members{...}
  - permet d'ajouter des variables d'instance, de classe, des méthodes
- Listes des règles
  - la génération de la classe ajoute les méthodes correspondant aux règles

Evaluation des AST - 1

```
i = INT { $v = Double.parseDouble(i.getText()); }
```

- i est un noeud particulier (CommonTree)

```
^(PLUS a = expr b = expr) { $v = $a.v + $b.v; }
```

- a = expr
  - expr est évaluée à partir du noeud courant (premier fils de PLUS, et renvoie une valeur indiquée par `$a.v`.
- b = expr
  - expr est évaluée à partir du noeud courant (deuxième fils de PLUS) et renvoie une valeur indiquée par `$b.v`.
- v est calculé.

Evaluation des AST - 2

```
^(PLUS . b = expr) { $v = $b.v; }
```

- . représente le premier fils de l'arborescence PLUS ; le noeud n'est pas évalué.
- b = expr : expr est évaluée à partir du noeud courant (deuxième fils de PLUS) et renvoie une valeur indiquée par `$b.v`.
- v est calculé.

```
{$v = 0;} ^( PLUS (a = expr { $v += $a.v ; } )+ )
```

- v est mis à 0
- Structure : arbre avec n fils : `^( PLUS (expr)+ )`
- expr est évaluée sur chaque fils de l'arbre et la valeur renvoyée affectée à `$a.v` incrémente v.

Sommaire

Génération d'arbre

- La grammaire AntLR suivante :

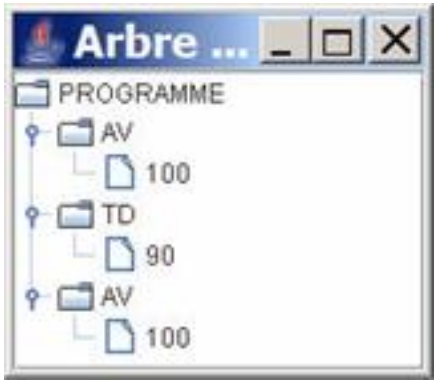
```
programme : liste_instructions
    -> ^(PROGRAMME liste_instructions)
;
liste_instructions : (instruction)+
;
instruction :
   ( AV^ | TD^) INT
;
```

- permet de générer un arbre AST

AST AntLR

A partir du programme :

```
AV 100
TD 90
AV 100
```



l'arbre AST suivant est généré :

```
(PROGRAMME (AV 100) (TD 90) (AV 100))
```

dont la représentation AntLR sous forme de liste est :

```
PROGRAMME DOWN AV DOWN 100 UP TD DOWN
90 UP AV DOWN 100 UP UP
```

qui utilise deux tokens imaginaires : UP et DOWN pour indiquer les niveaux dans l'arbre.

Fonctionnement du parser d'arbre

```
prog : ^(PROGRAMME (instruction)*) ;
instruction :
    ^(AV a = INT)
  | ^(TD a = INT) ;
```

- Chaque règle de la grammaire permet de générer une méthode du parser, ici sans paramètre, mais qui doit travailler sur une portion de l'arbre.
- L'algorithme du parser d'arbre utilise une pile où se trouvent les index des nœuds de l'arbre.
- Lors de son exécution, chaque méthode commence par dépiler l'index du premier token de l'arborescence à analyser et finit en empilant le nœud initial de la méthode suivante.

Simulation

```
PROGRAMME DOWN AV DOWN 100 UP TD DOWN
90 UP AV DOWN 100 UP UP
```

Grammaire : `prog : ^(PROGRAMME (instruction)*) ;`

1. prog dépile index(PROGRAMME)
2. prog empile index(AV)
3. instruction dépile index(AV)
4. instruction empile index(TD)
5. instruction dépile index(TD)
6. instruction empile index(AV)

Sommaire

Instruction répète

```
REPETE                                repeat  :
  |                                    ^(REPETE n = atom list_evaluation )
  --- 4                               ;
  |
  --- [
      |
      --- AV
            |
            --- 100
```

- La règle `atom` retourne un entier (r dans la structure).
- On considère que l'on a une autre règle `list_evaluation` permettant de parser un arbre dont la racine est le token `LIST` et dont les fils sont des instructions.

Instruction répète

- Il s'agit de répéter n fois les instructions dans l'arborescence second fils de l'arbre `REPETE`.
- Le parsing de la liste ne doit pas se faire par défaut d'où le symbole point (.) suivant `atom`.

```
repeat  :
 ^(REPETE n = atom . )
;

list_evaluation :
 ^(LIST instruction *)
;
```

Méthodes du parser d'arbre

- Le parser utilise un objet `input` de type `TreeNodeStream` (`CommonTreeNodeStream`).
- `mark()` : méthode qui retourne l'index du prochain symbole à lire.

Principe

- Le parsing différé se fait à l'aide d'une règle que l'on écrit dans la grammaire.
- On mémorise l'index sur lequel cette règle doit commencer le matching.
- Avant d'appeler la méthode du parser associée à la règle on empile l'index sur lequel elle doit commencer le matching et on dépile ensuite le nœud qu'elle aura placé pour la suite du matching et qui s'avère inutile.

Instruction répète

- ^(REPETE n = atom .) `{action}`
- arbre AntLR : `REPETE DOWN <atom> <.> UP`
- Il faut mémoriser l'index de l'arborescence représentée par le méta-symbole `.` que la règle `list_evaluation` doit connaître et qu'elle va dépiler lorsqu'elle est appelée.
- Dans le code de `{action}` il faut :
  - empiler l'index du nœud racine traité par `list_evaluation`.
  - appeler la méthode `list_evaluation()`
  - dépiler le nœud superflu que `list_evaluation()` va empiler.
  - Les méthodes `push` et `pop` de `LogoTree` allègent l'écriture.

Action

```
repeat
@init{
  int mark_list = 0;
}
  :
^(REPETE
   n = atom {mark_list = input.mark();} . )
  {
  // r est la variable résultat de atom.
   for (int i = 0; i < $n.r ; i++) {
     push(mark_list);
     list_evaluation();
     pop();
   }
  }
  ;
```

Méthodes

```
@members{
 public void push(int index) {
   ((CommonTreeNodeStream)input).push(index);
 }
 public void pop() {
   ((CommonTreeNodeStream)input).pop();
 }
}
```

Sommaire

Identificateurs

- Une table des symboles est une structure regroupant les informations sémantiques des identificateurs d'un programme :
  - Variable.
  - Type, Classe.
  - Méthode, fonction.
- Il est possible de construire plusieurs tables de symboles.
- Un dictionnaire de symboles est en général implémenté comme une table de hachage dont les clés sont les noms des symboles et les valeurs sont des structures déterminant les propriétés du symbole.
- La recherche d'un symbole est plus rapide dans une table de hachage.

Propriétés

- Les symboles ont au moins en commun les propriétés suivantes :
  - Un nom ; en général composé de lettres, de chiffres et quelques symboles particuliers comme _. Dans certains cas on peut avoir des signes d'opérateurs (+).
  - Une catégorie : variable, fonction, label, classe, etc.
  - Un type : il peut être défini dynamiquement (Python) ou statiquement (Java, C++).
- On peut représenter chaque catégorie par une classe séparée ayant comme attribut commun le nom et le type.

Erreurs

- Les tables de symboles permettent de déceler des erreurs :
  - Variable utilisée et non initialisée.
  - Procédure non déclarée.
  - Arité d'une procédure non respectée.
- Il est préférable de déceler au plus tôt de telles erreurs.

Implémentation

- Il est préférable de mettre le moins de code possible dans une grammaire.
- Utiliser des classes qui encapsulent le comportement des tables de symboles.e
- Instancier de telles classes et utiliser l'injection de dépendances pour les associer les instances avec les parsers.

Exemple

```
private void runParser() {
 try {
  // lexer
  LogoContext context = new LogoContext();
  ...
  LogoParser parser = new LogoParser(tokens);
  parser.setContext(context);
  LogoParser.programme_return r = parser.programme();
  ...
  LogoTree treewalker = new LogoTree(nodes);
  treewalker.setContext(context);
  treewalker.prog();
  ...
  } catch(Exception ex) {
ex.printStackTrace();
  }
}
```

Portées

- La portée est une région du code très bien délimitée qui groupe des définitions de symboles.
- Par exemple une portée de type fonction groupe les paramètres et les variables locales.
- Portée lexicale ou statique : une portée limitée par une paire de symboles particuliers : { . . . }

Particularité des portées

- Portée statique (la plupart des langages) vs portée dynamique (lisp).
  - portée dynamique : une fonction peut partager les variables de la fonction qui l'a invoquée.
- Beaucoup de portées ont des noms : fonction, classe.
- Les portées sont en général imbriquées : blocs de code.
- Certaines portées permettent des déclarations, des instructions.
- Les symboles dans une portée peuvent être visibles ou non à partir d'autres sections : les champs d'une classe ont des modificateurs de visibilité.

Portées imbriquées

```
int x;          // portée globale
void f() {      // f dans portée globale
  int j;        // j dans la portée f
  {int i;}      // i dans une portée imbriquée
  {int k;}      // k dans une autre portée imbriquée
}
void g() {      // g dans protée globale
  int i;        // i dans la portée g
}
```

Contexte

- Au moment du parsing :
  - on représente une portée par une structure qui contient le dictionnaire des symboles qui y sont déclarés.
  - la définition d'un symbole dans la portée revient à ajouter ce symbole dans le dictionnaire.
  - représenter des portées imbriquées revient à empiler les structures représentant les portées.
- Le contexte d'un symbole correspond à la portée dans laquelle il se trouve utilisé plus éventuellement les portées où elle est imbriquéee.

Opération sur un contexte

- Empilement d'une portée.
- La portée courante est la portée en sommet de pile.
- Définition d'un symbole dans la portée courante.
- Dépilement de la portée courante.

Portées imbriquées - 1

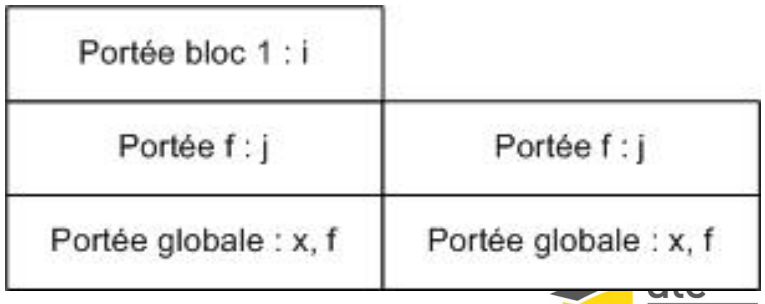
```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une portée imbriquée 2
}
void g() {    // 6. g dans protée globale
  int i;      // 7. i dans la portée g
}
```

- 1. : Empilement de la portée globale
- 1.1 : Définition de x dans la portée courante
- 2 : Définition de f dans la portée courante
- 2.1 : Empilement de la portée f

Portées imbriquées - 2

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une portée imbriquée 2
}
void g() {    // 6. g dans la portée globale
  int i;      // 7. i dans la portée g
}
```

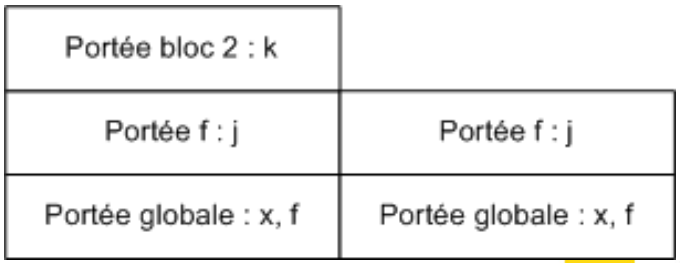
- 3. : Définition de j dans la portée courante (f)
- 4. : Empilement de la portée bloc 1
- 4.1 : Définition de i dans la portée courante
- 4.2 : Dépilement de la portée courante



Portées imbriquées - 3

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une portée imbriquée 2
}
void g() {    // 6. g dans portée globale
  int i;      // 7. i dans la portée g
}
```

- 5. : Empilement de la portée bloc 2
- 5.1 : Définition de k dans la portée courante
- 5.2 : Dépilement de la portée courante



Portées imbriquées - 4

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;}    // 4. i dans une portée imbriquée 1
  {int k;}    // 5. k dans une autre portée imbriquée 2
}
void g() {    // 6. g dans portée globale
  int i;      // 7. i dans la portée g
}
```

- 5.3 : Dépilement de la portée courante
- 6. : Définition de g dans la portée courante
- 6.1 : Empilement de la portée g
- 7. : Définition de i dans la portée (g)
- 7.1 : Dépilement de la portée (g)

Recherche d'un identificateur dans un contexte

- Un contexte possède une pile de portées.
- L'insertion d'un identificateur se fait dans le dictionnaire de la portée en sommet de pile.
- La recherche d'un identificateur se fait d'abord dans la portée courante.
- Si l'identificateur n'est pas trouvé, la recherche continue dans la portée sous le sommet de la pile et ainsi de suite jusque la portée où l'identificateur est trouvé.
- Si l'identificateur n'est trouvé dans aucune portée, l'identificateur n'a pas été défini.
- Si l'identificateur a été défini plusieurs fois, la recherche donne un résultat dans la portée la plus récente.

Recherche de x pour printf(x)

```
int x;        // 1. portée globale
void f() {    // 2. f dans portée globale
  int j;      // 3. j dans la portée f
  {int i;     // 4. i dans une portée imbriquée 1
   printf(x);
  }
  {int k;}    // 5. k dans une autre portée imbriquée 2
}
void g() {    // 6. g dans la portée globale
  int i;      // 7. i dans la portée g
}
```