ANTLR

Un compilateur de compilateurs

Introduction

- Un outils pour la reconnaissance de langages
  - Ecrit par Terence Parr en Java
- Plus facile d'utilisation que la plupart des outils similaires
- Propose un outils de mise au point
  - ANTLRWorks
    - Editeur graphique de grammaire et debuggeur
    - Ecrit par Jean Bovet
- Utilisé pour implémenter
  - De vrais langages de programmation
  - Des langages spécifiques à un domaine
    - Ex : fichiers de configuration
- Pour tout savoir : http://www.antlr.org
  - ANTLR + ANTRLWorks
  - Libre et open source

Introduction

- Compilateur de compilateur
  - Fichier source
    - Une description du langage
      - Grammaires EBNF
    - Éventuellement annotée
      - Association de différents types d'opérations aux règles de la grammaire
  - Produit
    - Le code pour l'analyseur lexical
    - Le code pour l'analyseur syntaxique
  - Le code produit peut être intégré dans les applications

- Supporte plusieurs langages cible
  - Java, Ruby, Python, C#, C et bien d'autres…

Introduction

- Supporte les grammaires LL(*)
  - Ne supporte que les grammaires récursives à droite
  - LL(k) : Regarde k lexèmes en avance pour lever des ambigüités
  - LL(*) : Regarde autant de lexèmes que nécessaire pour lever les ambigüités
- Avantage du LL(k)
  - Améliore la lisibilité des règles de la grammaire
  - Simplifie la description
  - Simplifie l'ajout de nouvelles règles

Introduction

- Trois grands types d'utilisation

1. Implémentation de « validateurs »
   - Valident si un texte en entrée respecte une grammaire
   - Pas d'action spécifique effectuée
2. Implémentation de « processeurs »
   - Valide un texte en entrée
   - Effectue les actions correspondantes
   - Les actions peuvent inclure
     - Des calculs (langages de script ou avoisinés)
     - De la mise à jour de base de données
     - L'initialisation d'une application à partir d'un fichier de configuration
     - …
3. Implémentation de « traducteurs »
   - Valide un texte
   - Le transforme dans un autre format
     - Ex : Java vers C++

Etapes de développement avec ANTLR

- Ecriture d'une grammaire

- Tester de debugger la grammaire sous ANTLRWorks
  - Offre la visualisation
    - d'arbres de dérivation syntaxique
    - d'arbres de syntaxe abstraits
  - L'exécution pas à pas de l'analyse d'un texte
  - Supporte la cible Java par défaut
    - i.e. vous pouvez inclure du code Java dans vos grammaires !

- Générer les classes depuis la grammaire
  - L'analyseur lexical et l'analyseur syntaxique

- Ecriture de l'application qui utilise les classes générées

- **Remarque** : dans les exemples et dans les TP, le langage Java sera utilisé

Structure d'un fichier de grammaire

- **grammar** *nom-grammaire* **;**
  - nom-gramaire : Nom donné à la grammaire
  - Doit correspondre au nom du fichier contenant la grammaire avec l'extension « .g »
- *options-de-grammaire* [optionnel]
  - Options permettant de décrire ce qui doit être généré par ANTLR
- *attributs-methodes-lexer-parser* [optionnel]
  - Ensemble des attributs et méthodes ajoutés aux classes d'analyseurs lexical et syntaxique
- *lexemes*
  - Ensemble des lexèmes utilisés dans la grammaire
- *regles*
  - Ensemble des règles définissant la grammaire

Les options des grammaires

- Syntaxe :

  option
  {
  name = *value* ;
  }

- Où *name* peut (principalement) prendre les valeurs suivantes
  - backtrack
    - Si backtrack = true : la grammaire est considérée comme LL(*)
  - language
    - Permet de signaler dans quel langage les classes doivent être générées
      - Ex : language = java, les classes sont générées en Java
  - output
    - La valeur associée à output définit le type de structure de donnée générée par l'analyseur syntaxique
      - Ex : output = AST, l'analyseur syntaxique génère un arbre syntaxique abstrait
  - k
    - Signale le nombre de lexèmes à utiliser pour choisir les règles
      - Ex : k = 1, signale que la grammaire est LL(1)

Ajout d'informations pour les classes et fichiers générés

- Ajout d'un en-tête au fichier généré
  - @lexer::header { … }
    - Signale des instructions à ajouter à l'en-tête du fichier contenant l'analyseur lexical
  - @parser::header { … }
    - Signale des instructions à ajouter à l'en-tête du fichier contenant l'analyseur syntaxique
  - Utile pour des inclusions de classes ou définition de package par exemple

- Ajout de méthodes / attributs
  - @lexer::members { … }
    - Fournit des méthodes / attributs définis par l'utilisateur pour l'analyseur lexical
  - @parser::members { … }
    - Fournit des méthodes / attributs définis par l'utilisateur pour l'analyseur syntaxique
  - Utile pour la factorisation de traitements pour les analyseurs ou encore pour ajouter des attributs permettant de partager des informations entre règles

Définition des lexèmes

- La syntaxe de définition des lexèmes est la suivante
  - *NomLexeme* **:** *expression-regulière* **;**
- Où
  - NomLexeme est l'identifiant d'un lexème
    - Cet identifiant DOIT commencer par une majuscule
  - expression-regulière est une expression régulière décrivant les mots associés au lexème
    - **'x'** : le caractère 'x'
    - **'a'..'t'** : l'ensemble des caractères dans l'intervalle ['a'; 't']
    - a**|**b : décrit l'alternative entre a et b
    - **.** : n'importe quel caractère
    - **~** : tous les caractères saufs ceux à droite de ~
    - **( )** : sert à regrouper des éléments
    - **+** : répétition 1 à n fois de l'élément à gauche de +
    - ***** : répétition 0 à n fois de l'élément à gauche de *
    - **?** : rend optionnel l'élément à gauche de ?

Définition des lexèmes

- Exemples
- Lexème reconnaissant un identifiant
  - ID : ('a'..'z'|'A'..'Z'|'_') ('a'..'z'|'A'..'Z'|'0'..'9'|'_')* ;
- Lexème reconnaissant un commentaire sur une ligne
  - COMMENT : '//' ~('\n'|'\r')* '\r'? '\n' ;
- Lexème reconnaissant un commentaire sur plusieurs lignes
  - COMMENT2 : '/*' .* '*/' ;
  - Attention : fonctionnement particulier du .*
    - Normalement, reconnait toute séquence de caractère
      - On ne s'arrête pas avant la fin du fichier
    - Mais avec ANTLR
      - On sort de la séquence dès que ce qui vient après est reconnu
      - Donc, dans l'exemple, dès que '*/' est trouvé
      - Pratique ☺

Définition des lexèmes

- Par défaut, l'analyseur lexical possède deux canaux de sortie pour les lexèmes
  - DEFAULT : le canal de sortie par défaut. Le lexème est consommé par l'analyseur syntaxique
  - HIDDEN : lorsqu'un lexème est sorti sur ce canal, il n'est pas traité par l'analyseur syntaxique
    - Utile pour les commentaires par exemple

- Lorsqu'un lexème est reconnu, une action peut-être effectuée avant que ce dernier soit transmis à l'analyseur syntaxique
  - Syntaxe :

    Lexeme **:** *expression-reguliere* **{** actions **} ;**

Définition des lexèmes

- Exemples d'actions
  - $channel = HIDDEN ;
    - Provoque la sortie sur le canal HIDDEN. Le lexème n'est donc pas traité par l'analyseur syntaxique
  - skip() ;
    - Le lexème qui vient d'être reconnu est simplement omis

- Retour sur l'exemple des commentaires :

```
COMMENT :  '//' ~('\n'|'\r')* '\r'? '\n'  {$channel=HIDDEN;}
   |   '/*' ( options {greedy=false;} : . )* '*/' {$channel=HIDDEN;}
   ;
```

  - Force les commentaires à être ignorés

Définition des règles

- Les règles sont décrites via la syntaxe EBNF inspirée par les expression régulières

- Syntaxe :

  regle : *corps-regle* ;

- Où
  - *regle* est le nom de la règle
    - Commence forcément par une minuscule
  - *corps-regle* est une expression décrivant ce qui doit être reconnu

Définition des règles

- Syntaxe du corps de la règle
  - *identifiant*
    - Si commence par une majuscule, réfère à un lexème préalablement déclaré qui doit être reconnu
    - Si commence par une minuscule, réfère à une règle qui doit être reconnue
  - ‘chaine’ : reconnait le mot ‘chaine’
    - Utilisé pour les mots clefs du langage par exemple
    - Ajoute « automatiquement » un lexème prioritaire
  - a **|** b : reconnait soit la sous-règle a, soit la sous règle b
  - a* : reconnait 0 à n fois la sous-règle a
  - a+ : reconnait 1 à n fois la sous-règle a
  - a? : rend optionnelle la reconnaissance de la sous règle a
  - () : sert à regrouper des sous-règles

Définition de règles

- Exemple de définition de règles

// Declaration de méthode à la C++

methodDecl : (*primitiveType* | *ID*) *ID* '(' (*varDecl* (',' *varDecl*)*)? ')' 'const'? ';' ;

// Types primitifs autorisés

primitiveType : 'char' | 'int' | 'float' | 'double' | 'bool' ;

// Déclaration de variable / paramètre

varDecl : (*primitiveType* | *ID*) *ID* ;

Définition de règles

- Vu d'ANTLR, une règle correspond à une méthode de l'analyseur syntaxique

- Elle peut posséder
  - Des paramètres
  - Une valeur de retour
  - Du code à exécuter
    - Au début de la règle
    - En fin de règle
    - À n'importe quel stade de l'analyse (en cours de règle)

Définition et utilisation des règles

- Syntaxe des règles

  regle **[** Type1 param1, Type2 param2**] returns [**TypeR1 id1, TypeR2 id2**]**
  @init { /* code en initialisation de règle */ }
  @after { /* code en fin de règle */ }
  : *corps-règle* { /*code en cours de règle */ } ;

- Manipulation des paramètres et valeurs de retour
  - $param1 : valeur du paramètre1
  - $id1 : valeur de retour

- Utilisation des paramètres et valeurs de retour
  - *regle2* : ret=*regle*[p1,p2] ;
  - p1 et p2 sont les paramètres
  - ret designe une structure contenant les valeurs de retour de *regle*
  - $ret.id1 et $ret.id2 désignent les deux valeurs de retour de *regle*

Exemple : la calculatrice en notation polonaise inverse en ~20 lignes

```
grammar calculatrice;

@parser::members
{ java.util.Stack<Integer> m_stack = new java.util.Stack() ; }

INT :  '0'..'9'+ ;

WS  :  (' ' | '\t' | '\r' | '\n') { $channel = HIDDEN ; } ;

expression
@after
{ System.out.println("Value on top: "+ m_stack.pop()) ; } :
        ( v = value { m_stack.push($v.val) ; }
        | v = operation[m_stack.pop(), m_stack.pop()] { m_stack.push($v.val) ; }
        )* ';' ;

value returns [Integer val] : a=INT { $val = Integer.valueOf($a.text) ; } ;

operation [Integer v1, Integer v2] returns [Integer val] :
        ('+'  {$val = $v1 + $v2 ; }
        | '-' {$val = $v1 - $v2 ; }
        | '*' {$val = $v1 * $v2 ; }
        | '/' {$val = $v1 / $v2 ; }
        ) ;
```

Exemple : la calculatrice en notation polonaise inverse

Evaluation interactive des expressions

```
import java.io.*;
import java.util.Scanner;
import org.antlr.runtime.*;

public class Processor {
	public static void main(String[] args) throws IOException, RecognitionException {
		new Processor().processInteractive();
	}

	private void processInteractive() throws IOException, RecognitionException  {
		Scanner scanner = new Scanner(System.in);
		while (true) {
			System.out.print("calculatrice> ");
			String line = scanner.nextLine().trim();
			if ("quit".equals(line) || "exit".equals(line)) break;
			Integer result = processLine(line);
			System.out.println("Resultat: "+result) ;
		}
	}

	private Integer processLine(String line) throws RecognitionException {
		calculatriceLexer lexer = new calculatriceLexer(new ANTLRStringStream(line));
		calculatricParser tokenParser = new calculatriceParser(new CommonTokenStream(lexer));
		calculatriceParser.expression_return parserResult = tokenParser.expression(); // start rule method
		return parserResult.resultat ;
	}
}
```

Exemple : la calculatrice en notation polonaise inverse

Evaluation des expressions à partir d'un fichier

```
import java.io.*;
import java.util.Scanner;
import org.antlr.runtime.*;

public class Processor {
	public static void main(String[] args) throws IOException, RecognitionException {
		if (args.length == 1) { // name of file to process was passed in
				Integer result = new Processor().processFile(args[0]);
				System.out.println(result) ;
		}
		else { // more than one command-line argument
				System.err.println("usage: java Processor file-name");
		}
	}

	private Integer processFile(String filePath) throws IOException, RecognitionException  {
		FileReader reader = new FileReader(filePath) ;
		calculatriceLexer lexer  =  new calculatriceLexer(new ANTLRReaderStream(reader));
		calculatriceParser parser  = new calculatriceParser(new CommonTokenStream(lexer)) ;
		calculatriceParser.expression_return parserResult = parser.expression() ;
		return parserResult.resultat ;
	}
}
```

Attributs des règles

- Les règles possèdent des attributs accessibles via
  - $nomRègle.attribut, accès direct à l'attribut
  - a=nomRègle et $a.attribut, récupération valeur de retour et accès à l'attribut

- Attributs prédéfinis
  - $r.text : String, le texte reconnu par la règle
  - $r.start : Token, le premier token à reconnaitre
  - $r.stop : Token, le dernier token à reconnaitre
  - $r.tree : Object, l'arbre AST généré par la règle

- Attributs correspondant aux valeurs de retour
  - Si valRet est une valeur de retour
  - $r.valRet permet d'accèder à la valeur de retour de la règle r

Attributs des tokens

- Les tokens possèdent des attributs accessibles via
  - $NomToken.attribut, accès direct à l'attribut
  - a=NomToken et $a.attribut, récupération valeur de retour et accès à l'attribut

- Attributs prédéfinis
  - $t.text : String, le texte associé au token t
  - $t.line : int, le numéro de ligne du premier caractère du token (commence à 1)
  - $t.pos : int, la position de ce caractère sur la ligne (commence à 0)

Le code généré

- Lors de la compilation, deux fichiers sont générés :
  - Un « lexer » (analyseur lexical)
  - Un « parser » (analyseur syntaxique)

- Ces fichiers contiennent une classe associée à chaque analyseur

- Soit une grammaire nommée Grammar
  - La classe GrammarLexer contiendra le « lexer »
  - La classe GrammarParser contiendra le « parser »

Le code généré pour l'analyseur syntaxique

- Les valeurs de retour des règles
  - Une classe générée par règle
  - Cette classe contient un attribut publique par valeur de retour de règle
  - Une référence vers l'arbre de syntaxe abstrait (si présent)
    - Accessible via getTree()
  - Nom normalisé : nomRegle_return

- Une règle = une méthode
  - Paramètres équivalents aux paramètres de la règle

- Les lexèmes (tokens) sont listés
  - Constantes de type int
  - Correspondent au « type » du lexème
  - Portent le nom du lexème
  - Permettra de faire la correspondance pour les arbres de syntaxe abstraits

Arbres de syntaxe abstraits

- ANTLR propose des fonctionnalités pour la construction d'arbres de syntaxe abstraits

- Objectifs d'un arbre de syntaxe abstrait
  - Stocker les lexèmes pertinents
  - Encoder la structure grammaticale sans information superflue
  - Etre facile à manipuler

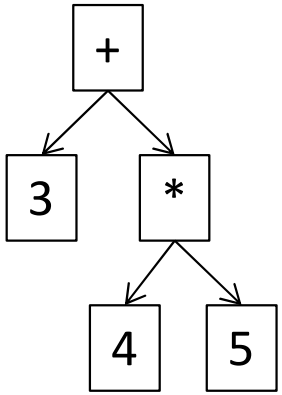
  ➢ *Il s'agit d'une structure encodant la sémantique du langage et simplifiant les traitements futurs*

Structure des arbres de syntaxe abstraits

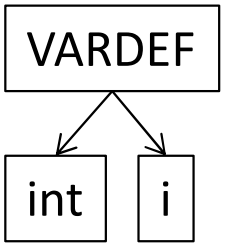
- En ANTLR un arbre de syntaxe abstrait :
  - Est composé de nœuds
  - Ces nœuds peuvent avoir de 0 à n fils
  - Les étiquettes associées aux nœuds sont des lexèmes
  - Rq : à partir des lexèmes, possibilité de récupérer le texte associé et donc de l'interpréter

- Construction d'un AST
  - ^(Token son1 son2)
    - Token est l'étiquette de l'AST
    - son1 et son2 sont des sous arbres
    - Rq : si son1 et / ou son2 sont des lexèmes, ils sont automatiquement convertis en AST
  - ^(ast1 son1 son2)
    - Construit un AST en ajoutant les sous arbres son1 et son2 à l'AST désigné par ast1

Exemples d'arbres de syntaxe abstraits



- Arbre correspondant à l'expression 3+4*5
- Encode la priorité des opérateurs
- Notation en ANTLR
  - ^('+' 3 ^('*' 4 5))



- Arbre correspondant à la déclaration d'une variable i de type int
- VARDEF est un lexème « imaginaire »
  - Non présent dans la grammaire
  - Ajouté pour donner une sémantique
- Notation en ANTLR
  - ^(VARDEF 'int' i)

Construction d'arbres de syntaxe abstraits à partir de règles de réécriture

- Manière recommandée de construire des arbres de syntaxe abstraits (AST)

- Notation :

  rule : « alt1 » **->** « créer-ceci-suivant-alt1 »
  | « alt2 » **->** « créer-ceci-suivant-alt2 »
  …
  | « alt2 » **->** « créer-ceci-suivant-alt2 »
  ;

- Remarques :
  - les règles peuvent générer 1 AST ou une liste d'AST
  - Par défaut : génération d'une liste d'AST correspondant aux éléments de la règle (si omission de l'opérateur ->)

Exemples simples

- stat: 'break' ';' -> 'break‘
  - Renvoie un AST composé d’un nœud avec l’étiquette ‘break’
  - Supprime le ‘;’

- decl : 'var' ID ':‘ type -> type ID ;
- type : ‘int’ | ‘float’
  - Renvoie deux AST contenant le type (int ou float) et l’identifiant
  - L’utilisation de type dans la production réfère à l’AST produit par la règle *type*
  - Rq : réordonne les éléments lus

- decl : ‘var’ ID ‘:’ type -> ^(VARDEF type ID)
  - Renvoie un AST avec comme racine le lexème VARDECL
  - Et comme fils l’AST renvoyé par type et un AST contenant le token ID

Les lexèmes ‘imaginaires’

- La règle précédente :
  - decl : ‘var’ ID ‘:’ type -> ^(VARDEF type ID)
  - Utilise un lexème nommé VARDEF
  - Ce lexème
    - n’a pas d’expression régulière associée
    - n’est pas présent dans la grammaire
    - Est ajouté pour donner de la **sémantique** à une opération

- Déclaration des lexèmes ‘imaginaires’
  - Sous la description des options, ajouter :

```
tokens
{
        VARDEF ;
        IM2 ; // Un autre lexème imaginaire…
}
```

AST et règles de réécriture

- Collecte d'une suite d'éléments
  - list : ID (',' ID)* -> ID+
    - Retourne la liste des AST correspondant aux identifiants ID
  - formalArgs : formalArg (',' formalArg)* -> formalArg+
    - Retourne la liste des AST retournés par la règle formalArg
  - decl : 'int' ID (',' ID)* -> ^('int' ID+) ;
    - Retourne un AST avec 'int' comme étiquette et un ensemble de fils correspondant à la liste des identifiants
  - compilationUnit : packageDef? importDef* typeDef+
    -> ^(UNIT packageDef? importDef* typeDef+) ;
    - Retourne un AST ayant comme étiquette UNIT
    - Un fils optionnel contenant l'AST retourné par *packageDef*
    - Une suite de 0 à n fils contenant les AST retournés par *importDef*
    - Une suite de 1 à n fils contenant les AST retournés pas *typeDef*

AST et règles de réécriture

- Duplication de nœuds et d'arbres
  - dup : INT -> INT INT ;
    - Retourne deux AST désignant le même lexème
  - decl : 'int' ID (',' ID)* -> ^('int' ID)+ ;
    - Retourne 1 à n AST ayant 'int' pour étiquette et un AST contenant le token ID comme fils
  - decl : type ID (',' ID)* -> ^(type ID)+ ;
    - Retourne 1 à n AST en duplicant l'AST retourné par type et en ajoutant ID comme fils
  - decl : modifier? type ID (',' ID)* -> ^(type modifier? ID)+ ;
    - Retourne 1 à n AST comme dans la règle précédente mais ajoute l'AST retourné par *modifier* comme sous arbre si celui-ci est présent

AST et règles de réécriture

- Choix des arbres en fonction de conditions
  - Possibilité d'ajouter des conditionnelles en tête des règles de réécriture
  - Choix de la règle en fonction du résultat de la conditionnelle

```
a[int which] : ID INT -> {which==1}? ID
                      -> {which==2}? INT
                      -> // Rien : pas d'arbre
;
```

AST et règles de réécriture

- Possibilité d'utiliser des labels dans les règles de réécriture
  - decl : 'var' i=ID ':' t=type -> ^(VARDEF $t $i)
    - i désigne la valeur de retour de ID
    - t désigne la valeur de retour de t
    - $i et $t sont utilisés pour construire l'arbre
  - prog: main=method others+=method*
    -> ^(MAIN $main) $others* ;
    - main désigne la valeur de retour de method
    - others est une liste des valeurs (opérateur +=) de retour de method*
    - Produit une liste d'AST, le premier ayant pour racine MAIN les autres les AST correspondant à method*

AST et règles de réécriture

- Règles de réécriture dans les sous-règles

```
Ifstat :
'if' '(' equalityExpression ')' s1=statement
('else' s2=statement  -> ^('if' ^(EXPR equalityExpression) $s1 $s2)
 |                    -> ^('if' ^(EXPR equalityExpression) $s1)
)
;
```

  - Retourne deux arbres différents en fonction de la présence de ‘else’
  - La présence du ‘else’ indique la règle de réécriture à utiliser

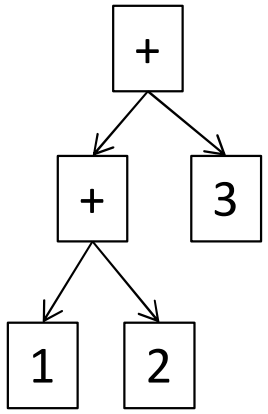
AST et règles de réécriture

- Parfois une simple réécriture n'est pas suffisante
  - Exemple

    expr : INT ('+' INT)*

    À l'analyse de 1+2+3 nous souhaitons générer

    ^('+' ^('+' 1 2) 3)



- Il faut construire l'arbre itérativement
  - expr : (INT -> INT) ('+' i=INT -> ^('+' $expr $i))* ;
    - Où $expr réfère au dernier AST construit dans la règle expr

Comment utiliser un AST

- Par défaut, ANTLR utilise
  - **org.antlr.runtime.tree.CommonTree** pour représenter les AST
  - **org.antlr.runtime.Token** pour représenter les lexèmes

- Lorsque les AST sont générés, les types de retour des règles contiennent une méthode getTree() renvoyant l'AST
  - Attention l'arbre est renvoyé via une référence sur Object
  - (CommonTree)valRet.getTree() pour convertir en CommonTree
  - Pas très beau mais possibilité d'inclure ses propres classes d'AST
    - Cf. livre sur ANTLR

    The Definitive ANTLR Reference: Building Domain-Specific Languages

    Terence Parr, ISBN: 978-0-9787-3925-6

org.antlr.runtime.tree.CommonTree

- Manipulation du lexème associé à l'arbre
  - Token getToken()
    - Récupération du lexème
  - int getType()
    - Récupération du type du lexème

- Manipulation des fils
  - int getChildCount()
    - Nombre de fils
  - Tree getChild(int i)
    - Fils à partir de son index
  - List getChildren()
    - Liste des fils
  - Tree getFirstChildWithType(int type)
    - Le premier fils d'un type donné

Conclusion

- ANTLR est un outils puissant
  - Description des grammaires LL(*) en EBNF
  - Paramètres et valeurs de retour pour les règles
  - Association de code aux règles
  - Gestion des arbres de syntaxe abstraits
  - Règles de réécriture
  - Simple d'utilisation étant donné sa puissance… ☺
- Ne faites plus d'analyse à la main
  - ANTLR est votre ami…
  - Fichiers de configuration, lecture de fichiers structurés
  - Interaction avec des langages de script
- Utilisez les arbres de syntaxe abstraite
  - Ex : multiplicité des langages de description de géométrie pour la 3D
  - Plusieurs analyseurs mais 1 seul AST… i.e. une seule phase de génération par analyse de l'AST

Pour aller plus loin…

The Definitive ANTLR Reference: Building Domain-Specific Languages

Terence Parr, ISBN: 978-0-9787-3925-6

- Une description complète de ANTLR
- Le contenu détaillé de cette partie du cours
- Autres informations :
  - Gestion des erreurs
  - Utilisation des patrons pour la réécriture
  - …