JavaCC et ANTLR : générateurs de parseurs

marco savard

Plan de la présentation

- Pourquoi «parser»?
- Théorie du «parsing»
- Diagramme syntaxique, BNF, grammaire
- Outils : JavaCC et ANTLR
- Quelques exemples
- Pièges à éviter et conseils

Pourquoi «parser»?

- Compilateur
- Interpréteur
- Rétro-ingénierie
- Audits automatique de code
- Métriques
- Langues naturelles (?)

Langages roulant avec la JVM

- Jacl (Tcl in Java)
- Jython (Python)
- JBasic
- Lisp
- Smalltak
- Cobol
- Ada
- Eiffel

- Une bonne centaines de langages : http://www.robert-tolksdorf.de/vmlanguages.html
- En utilisant Javacc/ANTLR pour parser et BCEL pour produire le bytecode, vous pourriez
- écrire votre propre langage de programmation (http://jakarta.apache.org/bcel/).

Expérience personnelle

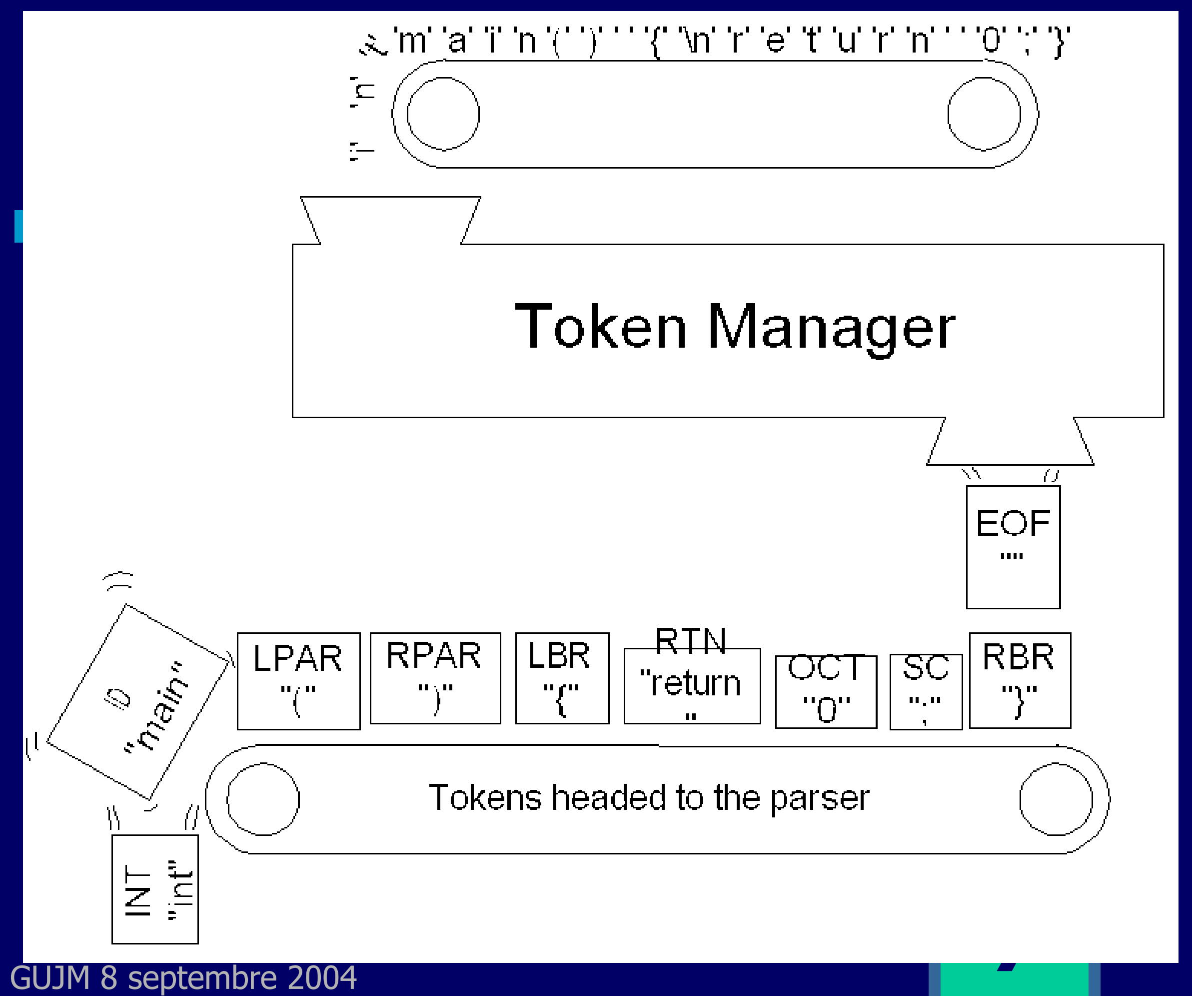
- Parseur de C++ avec lex/yacc
- Rétro-ingénierie de Java vers UML
- Définition et interprétation d'un langage propriétaire de script
- Audits et métriques de code Java

Théorie

- Analyse lexicale (lexer)
  - Erreur lexicale Java : un nombre mal construit (ex: 0x00fg, 1.3.2, 1.2e2.5)
- Analyse syntaxique (parser)
  - Erreur syntaxique Java : un modifiant non-permis (ex: abstract final void method())
- Analyse sémantique
  - Erreur sémantique Java : un méthode redéfinissante (overriding) qui ne propage (throws) pas une exception non-RunTime déclarée dans la méthode redéfinie (overridden) dans la super-classe.

Analyse lexicale (lexing)

Note : les commentaires sont consommés par le lexer



Analyse syntaxique (parsing)

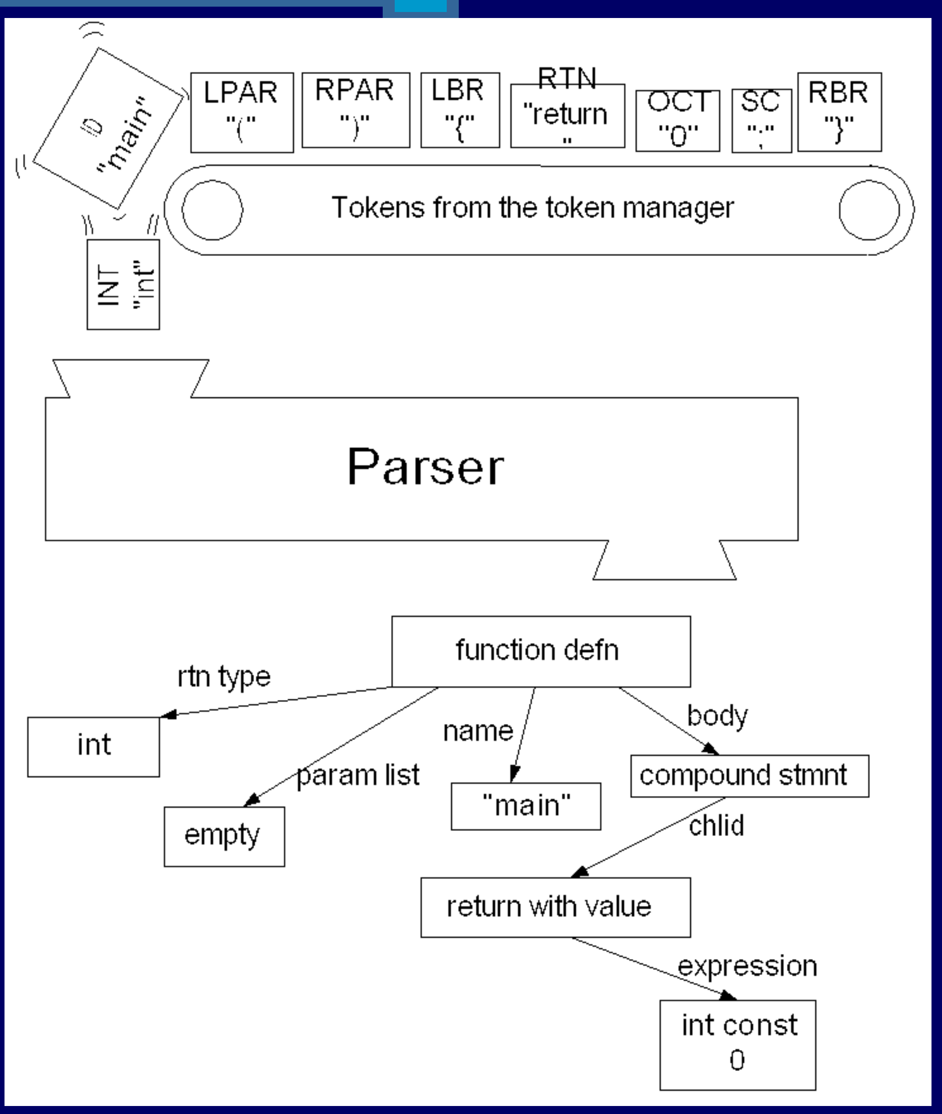
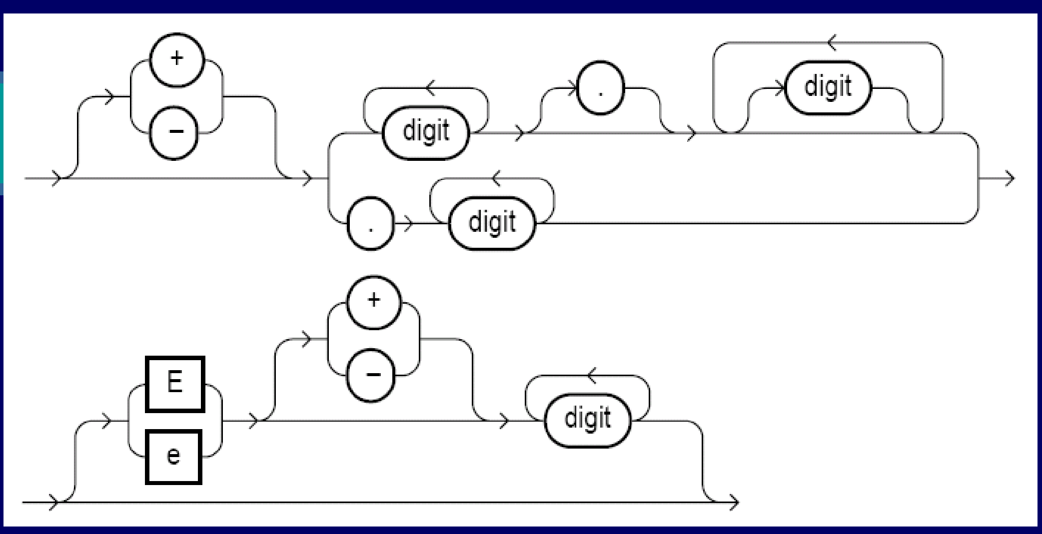
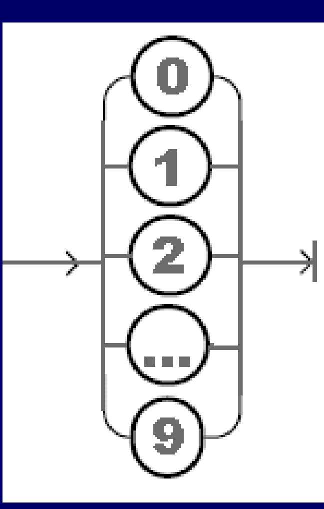


Diagramme lexicale

- Un nombre dans une BD Oracle :



Un “digit” :

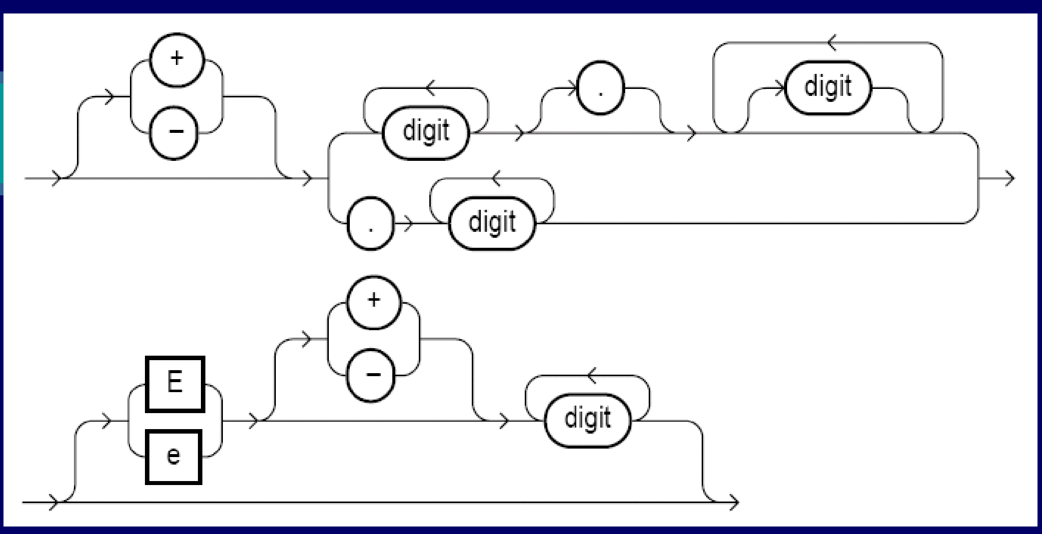


Est-ce valide?

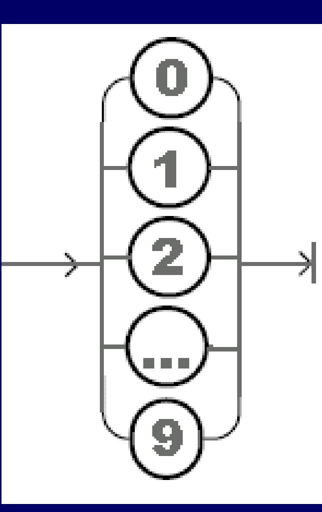
+32	0.5	070	1.0f	0.	10E2.5
0e3	.1	0xff	1e0	-0.	PI

Diagramme lexicale

- Un nombre dans une BD Oracle :



Un “digit” :



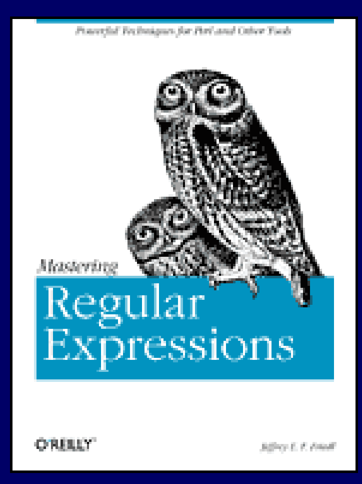
Expression régulière:

number :=

(+|-)? (([0-9]+ (.)? [0-9]*) | (. [0-9]+)) ((e|E) (+|-)? [0-9]+)?

Expressions régulières

- Regular expression ou “regex” en anglais
- Texte décrivant un patron de recherche
- Concept de “wildcards” étendu (ex: *.txt)
- Outils UNIX (grep, awk, Tcl, Perl..)
- Librairies depuis Java 1.4 (java.util.regex)



Pour plus de détails:

Mastering Regular Expressions, O'Reilly and Associates, 1997.

Tutorial Sun : **Regular Expressions**
http://java.sun.com/docs/books/tutorial/extra/regex/index.html

Expressions régulières

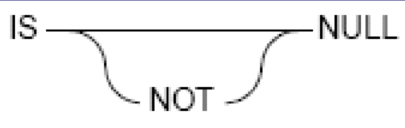
- KEYWORD

variable

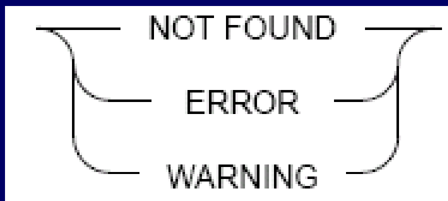
- [a-z]+

ADD Clause p. 3-288

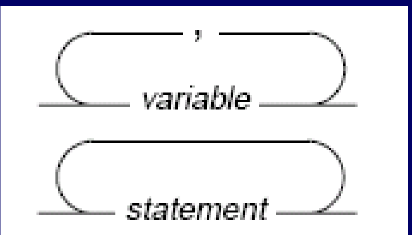
- add_clause



- IS (NOT)? NULL



- (NOT FOUND) | ERROR | WARNING



- variable (, variable)*
- (statement)+

Un nombre en Java

- **Java1_4.jj (JavaCC):**
- TOKEN :{
- < INTEGER_LITERAL:
- <DECIMAL_LITERAL> (["l","L"])?
- | <HEX_LITERAL> (["l","L"])?
- | <OCTAL_LITERAL> (["l","L"])?
- >
- | < #DECIMAL_LITERAL: ["1"-"9"] (["0"-"9"])* >
- | < #HEX_LITERAL: "0" ["x","X"] (["0"-"9","a"-"f","A"-"F"])+ >
- | < #OCTAL_LITERAL: "0" (["0"-"7"])* >
- **Java1_5.g (ANTLR):**
- NUM_INT :
- '.' ('0'..'9')+ (EXPONENT)? (FLOAT_SUFFIX)?
- | ('0' (('x'|'X') (HEX_DIGIT)+) | ('0'..'7')+)?
- | ('1'..'9') ('0'..'9')* ('l'|'L')

Backus-Naur Form (BNF)

- Formule mathématique pour décrire formellement un langage
- Utilisé par Backus pour décrire Algol60
- symbol := alternative1 | alternative2
  - LHS (Left-Hand Side) : symbol
  - RHS (Right-Hand Side): the rest
  - Règle terminale : où le RHS est formé uniquement de lexèmes (tokens)
  - Règle non-terminale : où le RHS contient des règles définies ailleurs par des LHS
  - Règle récursive : où un des éléments du RHS est dans le LHS
- Details : http://www.garshol.priv.no/download/text/bnf.html

Java grammar BNF

- CompilationUnit :=
-     OptPackageDeclaration
-     ImportDeclarations
-     TypeDefinitions
-     EOF

- OptPackageDeclaration :=
- PackageDeclaration |
- /*nothing */
- ;

- PackageDeclaration :=
-   "package" IDENTIFIER ";"
-
- ImportDeclarations :=
-   ImportDeclarations importDeclaration

- ImportDeclaration :=
- "import" IDENTIFIER ";"

- TypeDeficitions :=
-   TypeDefinition

- TypeDefinitions :=
- TypeDefinitions TypeDefinition

- TypeDefinitions :=
- ClassDefinition

- TypeDefinition :=
- InterfaceDefinition

Trouvez les LHS, RHS, règles terminales et non-terminales, etc.
Note : Les parseurs LR, tels que yacc, préfèrent la recursion a gauche (comme ImportDeclarations importDeclaration, voir [LEX&YACC] page 197).

Extended Backus-Naur Form (BNF)

- Regex + BNF = EBNF
- Support de
  - [optional] or (optional)?
  - (one_or_many)+
  - (zero_or_many)*

Java grammar EBNF

- Java1_4.jj (JavaCC) :

```
void CompilationUnit() :
{}
{
  [ PackageDeclaration() ]
  ( ImportDeclaration() )*
  ( TypeDeclaration() )*
  <EOF>
}

void PackageDeclaration() :
{}
{  "package" Name() ";" }

void ImportDeclaration() :
{}
{   "import" Name() [ "." "*" ] ";" }
{ hook_method() } …
```

Java1_5.g (ANTLR) :

```
compilationUnit :
  ( packageDefinition | /*nothing*/ )
  ( importDefinition ) *
  ( typeDefinition )*
  EOF!
  ;

packageDefinition :
  p:"package"  identifier SEMI!
   ;

ImportDefinition
  options { hook_method() }  :
  i:"import" ("static")? identifierStar SEMI!
  ;

typeDedinition :
  m:modifiers (classDefinition |
interfaceDefinition)
  ;
```

Outils

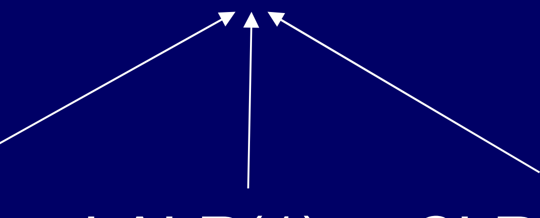
- Lex (GNUs' flex) : lexer
- Yacc (GNU's bison) : LALR(1) parser
- Jlex : Java lexer
- CUP : Java LALR(1) parser
- PCCTS -> ANTLR: LL(k) parser (C, puis Java)
- SableCC : LL(k) parser
- JavaCC : LL(k) parser

Topologie des parseurs

- L Parsers
  - **L**eft-to-right parsers

LR parsers
Left-to-right **R**ightmost-first parsers
(bottom-up parsers)

LL parsers
Left-to-right **L**eftmost-first parsers
(top-down parsers)



SLR
Simple LR

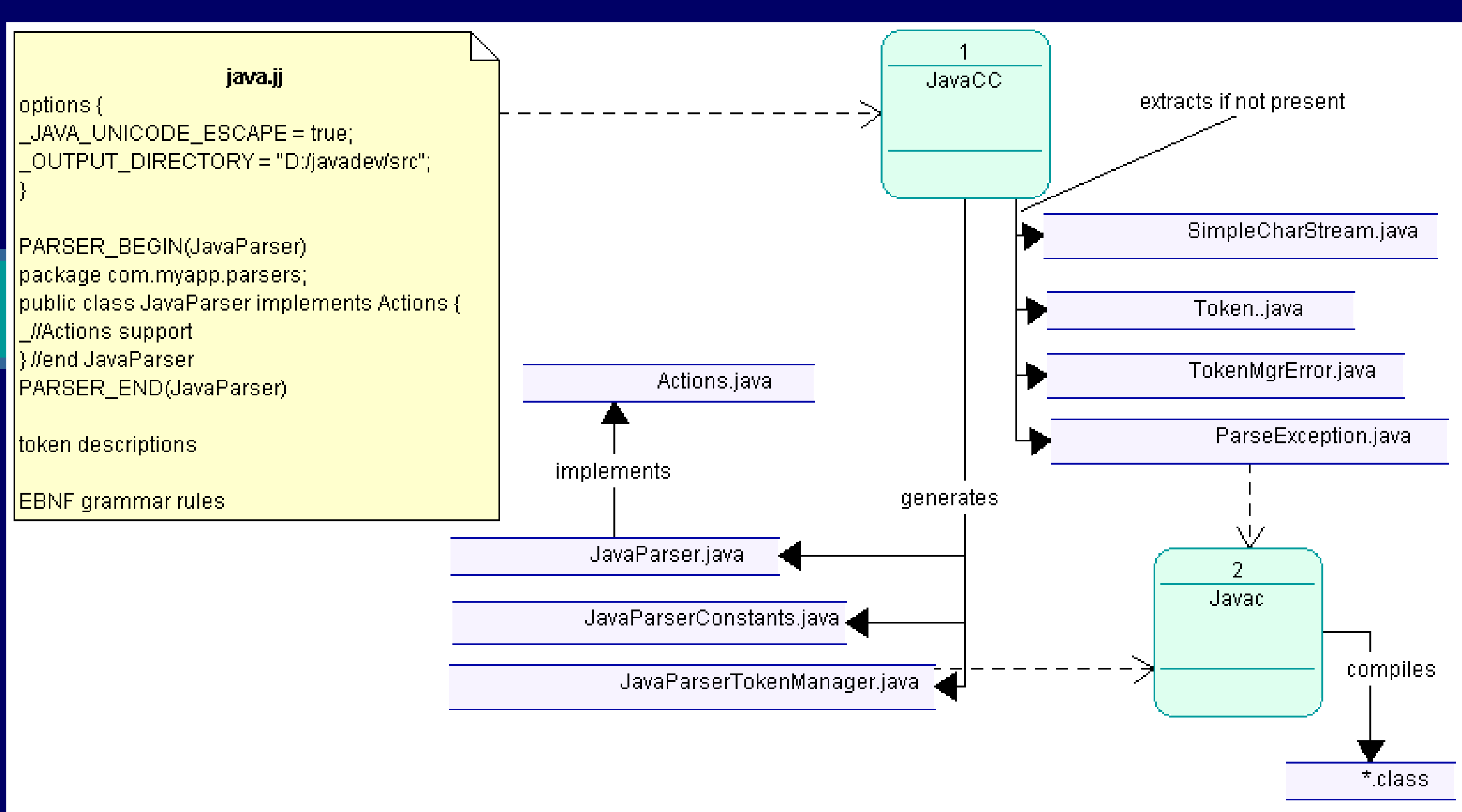
LALR(1)
Look-**a**head LR
(yacc, bison, CUP)

CLR
Canonical LR

LL(k)
ANTLR, JavaCC

Ref:http://encyclopedia.thefreedictionary.com/parser

“Workflow” de JavaCC



1) If JAVA_UNICODE_ESCAPE = true -> extracts JavaCharStream.java
2) JavaCC avant 3.0 => ASCII_CharStream.java & ASCII_UCodeESC_CharStream.java
3) Tout est genere dans le paquetage com.myapp.parsers

“Workflow” de ANTLR

java15.g

```
header {
package declaration;
import clauses;
}

class JavaParser extends Parser implements Actions;
options {
k = 2;      // two token lookahead
exportVocab=Java;  // generates the parser in Java
..
}

tokens {
..
}

grammar rules
```

1
ANTLR

Actions.java

implements

antlr.jar

JavaParser.java

JavaLexer.java

JavaTokenTypes.java

2
Javac

1) If exportVocab = Java, le parseur est généré en langage Java.
2) Tout est genere dans le paquetage spécifié dans la clause declaration;

Ambuiguité syntaxique

- **Grammaire ambiguë :**
- Expression :=
  - Expression '+' expression
  - Expression '*' expression
  - '(' expression ')'
  - NUMBER
  - ;
  - 2+3*4 : is ambiguous

Grammaire non-ambiguë:

```
Expression :=
   expression '+' mulexpr
   | mulexpr
   ;

Mulexpr :=
   mulexpr '*' primary
   | primary
   ;

Primary :=
   '(' expression ')'
   | NUMBER
   ;
```

Note : Java1_4.jj de JavaCC et Java1_5.g de ANTLR utilisent cette approche (AdditiveExpression and MultiplicativeExpression); yacc utilise une technique de priorité d'opérations (+ compliqué).

Répertoires de grammaires

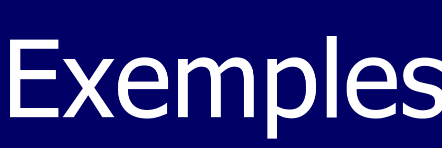
- JavaCC (*.jj files):
  - Ada
  - C/C++
  - HTML
  - IDL (Corba)
  - Java
  - Python
  - Rational Rose (Petal)
  - SQL (ANSI)
  - Visual Basic
  - VRML
  - XML
  - See:http://www.cobase.cs.ucla.edu/pub/javacc/

- ANTLR (*.g files):
  - Ada
  - C/C++/C#
  - HTML
  - IDL (Corba)
  - Java
  - OCL (UML constraints)
  - Pascal
  - Python
  - RCS files (CVS)
  - SQL (SQLServer2000, Oracle7)
  - VRML
  - See: http://www.antlr.org/grammar/list

Utilisation

- JavaCC:
  - Grammaire *.jj
  - Trois fichiers généres
  - Quatre fichiers extraits

ANTLR:
- Grammaire .g
- Trois fichiers générés
- Dépend d'une librarrie externe antlr.jar



Exemples

- Faire la démo JavaCC/ANTLR

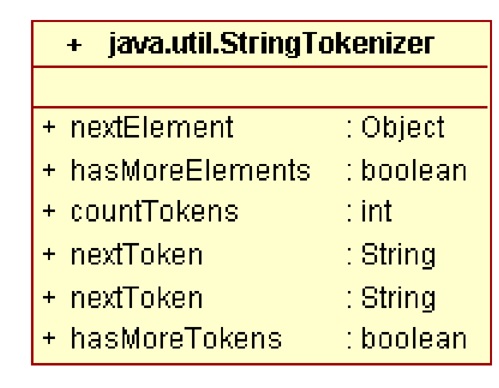
JavaCC versus ANTLR

- Avantages JavaCC :
  - Interface usager intuitif
  - Syntaxe de la grammaire plus simple
  - Parseur autonome (sans .jar)
  - (pour Java) grammaire plus tolérante
- Avantages ANTLR :
  - Historique «open source» plus long
  - Peut générer des parseurs en Java ou en C++
  - Intégration avec Eclipse
  - Un peu plus de grammaires disponibles

Alternatives aux parseurs

- StringTokenizer
- StreamTokenizer
- Interpretor Design Pattern
- SAX/DOM XML libraries

StringTokenizer



- Convient pour séparer des mots dans une chaîne
- Un seul type de séparateur
- Un seul type de token

StreamTokenizer

+ java.io.StreamTokenizer

Member	Type
- ttype	: int
- TT_EOF	: int
- TT_EOL	: int
- TT_NUMBER	: int
- TT_WORD	: int
- sval	: String
- nval	: double
+ toString	: String
+ nextToken	: int
+ pushBack	: void
+ resetSyntax	: void
+ wordChars	: void
+ whitespaceChars	: void
+ ordinaryChars	: void
+ ordinaryChar	: void
+ commentChar	: void
+ quoteChar	: void
+ parseNumbers	: void
+ eolIsSignificant	: void
+ slashStarComments	: void
+ slashSlashComments	: void
+ lowerCaseMode	: void
+ lineno	: int

- Convient pour séparer des tokens d’un flux entrant
- Quatre types de tokens par défaut
- Possibilité de rédéfinir les tokens

Librairies DOM/SAX

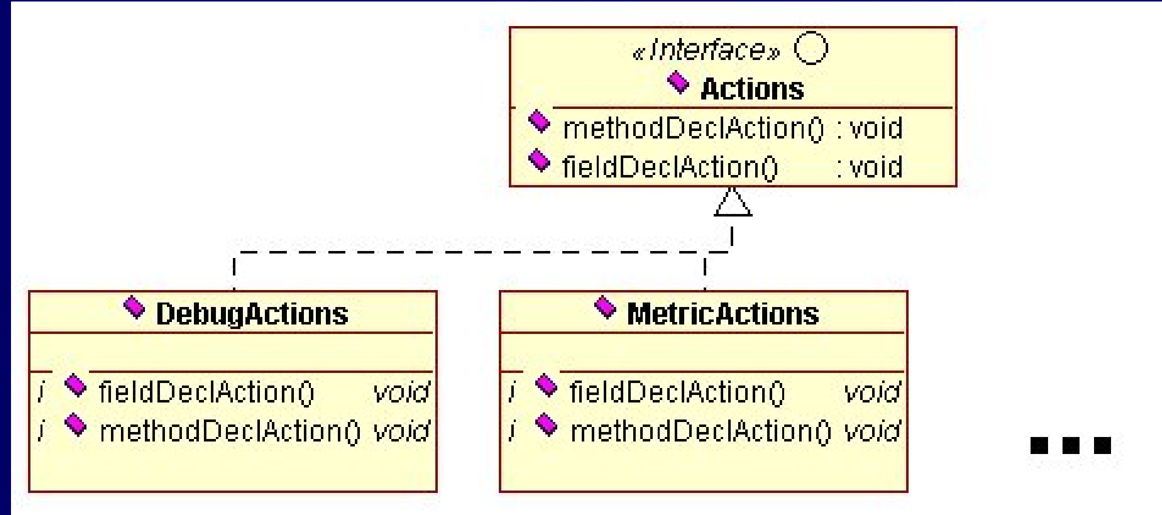
- Même si JavaCC et ANTLR ont des grammaires XML, vaut mieux utiliser DOM/SAX
- DOM préférable en général (mais plus lent).
- SAX est évenementiel (event-driven), donc plus rapide et plus près de l'approche des parseurs.

Pièges

- Ne pas définir de paquetage de génération
- Mélanger les classes générées de parsage et les classes normales dans le paquetage.
- Mettre trop de code dans la grammaire
- Les commentaires sont perdus à l`analyse lexicale (solution : lexer hook methods)

Conseils

- Définir une interface Actions.java, puis faites hériter des implémentations de Actions
- Définir une classe implémentant Actions.java pour le debogage (trace)
- Laisser LA = 1



Références

- LALR(1) :
  - [LEX&YACC] O'Reilly & Associates, 1995, 366 pages
- LL(k) :
  - Presque rien sur JavaCC ou ANTLR
  - Références électroniques seulement