



ANTLR4 Basics

Andrea Polini, Luca Tesei

Compilers
MSc in Computer Science
University of Camerino

What's that?

ANTLR v.4 is a **powerful parser generator** that you can use to read, process, execute, or translate structured text or binary files.

From a grammar as a formal language description, ANTLR generates a parser for that language that can automatically build parse trees. ANTLR also automatically generates tree walkers that you can use to visit the nodes of those trees to execute application-specific code.

What's that?

ANTLR v.4 is a **powerful parser generator** that you can use to read, process, execute, or translate structured text or binary files.

From a **grammar as a formal language description**, ANTLR generates a parser for that language that can automatically build parse trees. ANTLR also automatically **generates tree walkers** that you can use to visit the nodes of those trees to execute application-specific code.

How can I get it?

- Download last complete jar from <http://www.antlr.org/download.html>
- Put it in an appropriate folder, e.g. `/usr/local/lib`
- The jar contains:
 - all dependencies necessary to run the ANTLR tool
 - the runtime library needed to compile and execute recognizers generated by ANTLR
 - a sophisticated tree layout support library:
<http://code.google.com/p/treelayout>
 - a template engine useful for generating code and other structured text: <http://www.stringtemplate.org>

How can I install it?

- Set the CLASSPATH environment variable to include "." and the jar:

```
> export
```

```
CLASSPATH="./usr/local/bin/antlr-4.7.2-complete.jar:$CLASSPATH"
```

- You can do it every time you start a session in a shell or you can edit the `.bash_profile` file

- To run the ANTLR4 Tool:

```
> java -jar /usr/local/lib/antlr-4.7.2-complete.jar  
or directly:
```

```
> java org.antlr.v4.Tool
```

- To save typing:

```
> alias antlr4='java -jar /usr/local/lib/antlr-4.0-complete.jar'
```

How should I use it?

File Hello.g4

```
grammar Hello; // Define a grammar called Hello
r : 'hello' ID ; // Match the word 'hello' followed by an identifier
ID : [a-z]+ ; // Match lower-case identifiers
WS : [\t \r \n]+ -> skip ; // skip spaces, tabs, newlines, \r (Windows)
```

```
> antlr4 Hello.g4
```

produces:

```
Hello.g4 HelloLexer.java HelloParser.java
```

```
Hello.tokens HelloLexer.tokens
```

```
HelloBaseListener.java HelloListener.java
```

Then:

```
> javac *.java
```

Testing Hello

- ANTLR4 generates an executable recognizer embodied by `HelloParser.java` and `HelloLexer.java`
- There is not (yet) a main program to trigger language recognition
- ANTLR4 provides a flexible testing tool in the runtime library called `TestRig`
- ```
> alias grun='java org.antlr.v4.gui.TestRig'
```
- The test rig takes:
  - a grammar name
  - a starting rule name
  - various options for the desired output

# Testing Hello

```
> grun Hello r -tokens # start the TestRig on grammar Hello at rule r
hello parrt # input for the recognizer that you type
<eof> # type ctrl+D on Unix or ctrl+Z on Windows}\\
```

Outputs a detailed description of the tokens:

```
[@0,0:4='hello',<1>,1:0]
[@1,6:10='parrt',<2>,1:6]
[@2,12:11='<EOF>',<-1>,2:0]
```



# Testing Hello

```
> grun Hello r -tree
hello parrt
<eof>
```

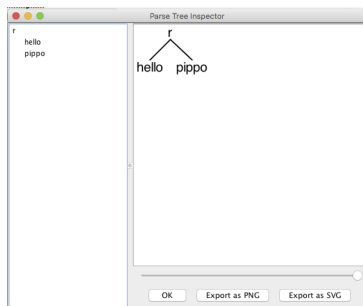
Outputs the parse tree in LISP-style text:

```
(r hello parrt)
```

# Testing Hello

```
> grun Hello r -gui
hello pippo
<eof>
```

Opens a graphical representation of the parse tree:



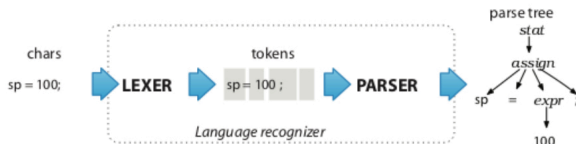
# ANTLR4 Plugin for IDEs

- For Eclipse: <https://github.com/jknack/antlr4ide>
- For other IDEs: <https://www.antlr.org/tools.html>

# Compiler Phases in ANTLR4

## Phases

ANTLR4 follows the usual conceptual structure of a generic compiler that we have seen in this course



# Grammars and Parsers in ANTLR4

## Grammar Definitions

Rules defines non-terminal symbols starting with lower-case letters

```
assign : ID '=' expr ';' ; // match an assignment statement like "sp = 100;"
```

## Grammar Implementation

ANTLR4 essentially creates a Recursive Descent Parser for the given grammar

```
// assign : ID '=' expr ';' ;
void assign() { // method generated from rule assign
 match(ID); // compare ID to current input symbol then consume
 match('=');
 expr(); // match an expression by calling expr()
 match(';');
}
```

# Lookaheads

## Lookaheads

ANTLR4 autonomously decide how many lookaheads are needed to take parsing decision (even the whole text!)

```
/** Match any kind of statement starting at the current input position */
stat: assign // First alternative ('|' is alternative separator)
 | ifstat // Second alternative
 | whilestat
 ...
 ;
```

## Left Recursion

ANTLR4 accepts left recursive grammars and handles them transparently!

```
void stat() {
 switch («current input token») {
 CASE ID : assign(); break;
 CASE IF : ifstat(); break; // IF is token type for keyword 'if'
 CASE WHILE : whilestat(); break;
 ...
 default : «raise no viable alternative exception»
 }
}
```

# Ambiguity

## Ambiguity

ANTLR4 accepts ambiguous grammars, but it cannot decide alone on which parse tree to generate for ambiguous sentences

```
stat: expr ';' // expression statement
 | ID '(' ')' ';' // function call statement
 ;
expr: ID '(' ')'
 | INT
 ;
```

f()); as expression



f()); as function call



# Ambiguity

- ANTLR4 will create, for an ambiguous sentence, the first parse tree that can be generated
- The order in which the rules are written in the .g4 file matters!
- In case of multiple choices the first rule is applied
- In case of fail, backtrack!

This resolves also possible ambiguities in LEXER (rules defining symbols starting with upper-case letters):

```
BEGIN : 'begin' ; // match b-e-g-i-n sequence; ambiguity resolves to BEGIN
ID : [a-z]+ ; // match one or more of any lowercase letter
```

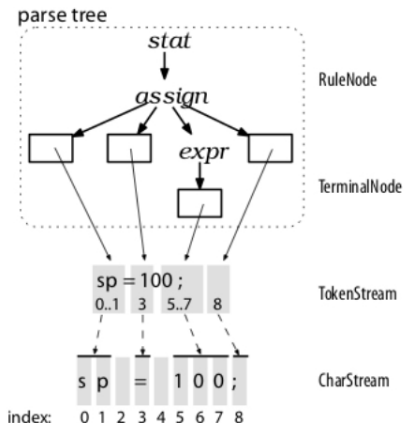


# Semantic Analysis and Code Generation

- ANTLR4 permits the definition of Syntax Directed Translation Schemes
- However, the **main and preferred** way to implement actions associated to parsing is through **walking** or **visiting** the generated parse tree
- This has a lot of advantages in modularity and re-usability

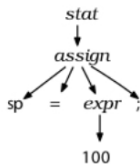
# ANTLR4 Java Classes

- ANTLR4 creates by default Java code for a given .g4 file
- Some ANTLR4 classes are CharStream, Lexer, Token, Parser and ParseTree

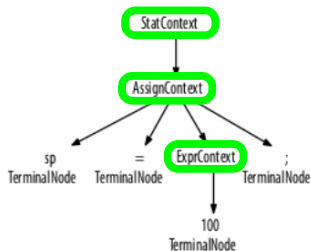


# ANTLR4 Java Classes for Rules

- ANTLR4 creates specific subclasses for each symbol
- This facilitates accessing to the subtrees



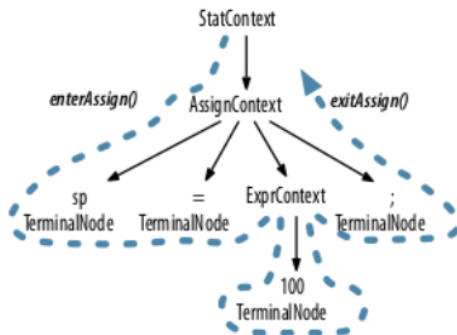
Parse tree



Parse tree node class names

# Run-time tree walking

- By default ANTLR4 generates a parse tree *listener* interface
- This responds to events triggered by the built-in tree walker
- The built-in tree walker performs a dept-first left-to-right visit of the parse tree
- For each node rule `name` two methods `enterName()` and `exitName()` are created:

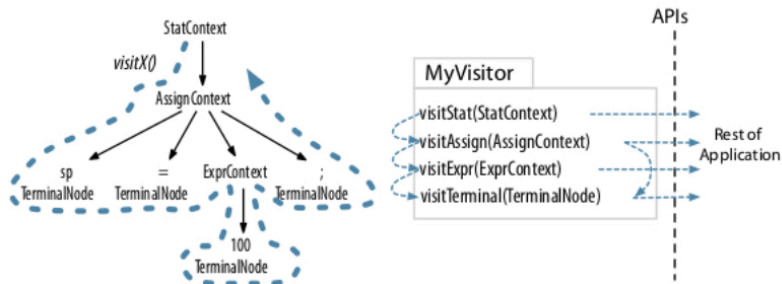


# Run-time tree walking



# Run-time tree visitors

- We can also decide a particular order in which the tree is visited, different from the standard one
- Call ANTLR4 with `-visitor` option
- It generates a visit method for each rule name
- Inside the code we have to make explicit calls to the other visit methods



# Starter Project

- Let's create the first application
- We want to parse integer lists inside possibly nested curly braces:  
{1, 2, 3} or {1, {2, 3}, 4 }
- We want to produce corresponding strings of Unicode characters
- E.g., {1, 2, 3} is translated to "\u0001\u0002\u0003"

```
starter/ArrayInit.g4
/** Grammars always start with a grammar header. This grammar is called
 * ArrayInit and must match the filename: ArrayInit.g4
 */
grammar ArrayInit;

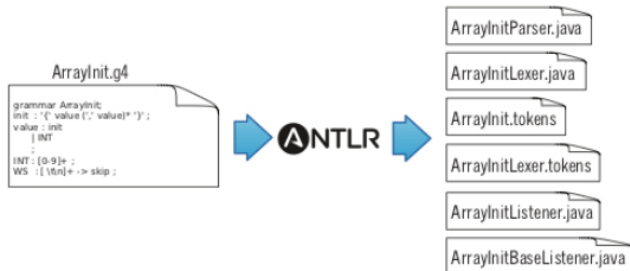
/** A rule called init that matches comma-separated values between {...}. */
init : '{' value (',' value)* '}' ; // must match at least one value

/** A value can be either a nested array/struct or a simple integer (INT) */
value : init
 | INT
 ;

// parser rules start with lowercase letters, lexer rules with uppercase
INT : [0-9]+ ; // Define token INT as one or more digits
WS : [\t\r\n]+ -> skip ; // Define whitespace rule, toss it out
```

# Starter Project

- Let's run ANTLR4 and produce the stub code:





# Starter Project

- Analyse the code
- Create simple Test class
- Create a subclass to define actions at enter and exit of the rules
- Create a class for realising the translation

# Expressions Project

- Let's create an ANTLR4 project for a desk calculator
- It will parse sequences of expressions and will print the corresponding value

```
tour/Expr.g4
Line 1 grammar Expr;
-
- /** The start rule; begin parsing here. */
- prog: stat+ ;
5
- stat: expr NEWLINE
- | ID '=' expr NEWLINE
- | NEWLINE
- ;
10
- expr: expr ('*' | '/') expr
- | expr ('+' | '-') expr
- | INT
- | ID
15 | '(' expr ')'
- ;
-
- ID : [a-zA-Z]+ ; // match identifiers
- INT : [0-9]+ ; // match integers
20 NEWLINE: '\r'? '\n' ; // return newlines to parser (is end-statement signal)
- WS : [\t]+ -> skip ; // toss out whitespace
```

# Importing grammars

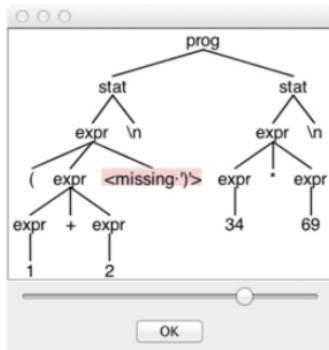
- ANTLR4 permits to import grammars
- Very useful for modularity

```
tour/LibExpr.g4
grammar LibExpr; // Rename to distinguish from original
import CommonLexerRules; // includes all rules from CommonLexerRules.g4
/** The start rule; begin parsing here. */
prog: stat+ ;
```

# Handling Errors

- ANTLR4 automatically handles errors
- The standard behaviour can be customized (advanced topic)

```
⇒ $ grun LibExpr prog -gui
⇒ (1+2
⇒ 34*69
⇒ E0
```



# Rule labeling

- When rules have alternatives it is better to give names to them

## tour/LabeledExpr.g4

```
stat: expr NEWLINE # printExpr
 | ID '=' expr NEWLINE # assign
 | NEWLINE # blank
 ;

expr: expr op=('*' | '/') expr # MulDiv
 | expr op=('+' | '-') expr # AddSub
 | INT # int
 | ID # id
 | '(' expr ')' # parens
 ;
```

# Calculator Implementation with Visitor

- Let's implement the calculator using the Visitor Pattern

⇒ `$ antlr4 -no-listener -visitor LabeledExpr.g4`

First, ANTLR generates a visitor interface with a method for each labeled alternative name.

```
public interface LabeledExprVisitor<T> {
 T visitId(LabeledExprParser.IdContext ctx); # from label id
 T visitAssign(LabeledExprParser.AssignContext ctx); # from label assign
 T visitMulDiv(LabeledExprParser.MulDivContext ctx); # from label MulDiv
 ...
}
```

# Calculator Implementation with Visitor

- Subclass `LabeledExprBaseVisitor<T>` with `T` as `Integer`
- Redefine the behaviour of the visit methods
- Create a class with a main that creates a visitor object and visits a parse tree
- See Code...

# Translator from Java classes to Java interfaces

- Let's implement a translator that can parse Java files!
- We are given a Java grammar specification `Java.g4`
- The translator has to transform the code of a Java class into a code for a Java interface containing the same methods without implementation
- Any comment appearing within the method signature must be retained

`tour/Java.g4`

```
classDeclaration
 : 'class' Identifier typeParameters? ('extends' type)?
 ('implements' typeList)?
 classBody
 ;
```

`tour/Java.g4`

```
methodDeclaration
 : type Identifier formalParameters ('[' ']')* methodDeclarationRest
 | 'void' Identifier formalParameters methodDeclarationRest
 ;
```



# Translator from Java classes to Java interfaces

```
tour/Demo.java
import java.util.List;
import java.util.Map;
public class Demo {
 void f(int x, String y) { }
 int[] g(/*no args*/) { return null; }
 List<Map<String, Integer>>[] h() { return null; }
}
```

must produce (see code):

```
tour/IDemo.java
interface IDemo {
 void f(int x, String y);
 int[] g(/*no args*/);
 List<Map<String, Integer>>[] h();
}
```

# Implementing an SDT in ANTLR4

- Let's implement a translator that parses a csv text file with tab as separator
- We want to select the data values of a particular column

tour/t.rows

```
parrt Terence Parr 101
tombu Tom Burns 020
bke Kevin Edgar 008
```

Base grammar:

```
file : (row NL)+ ; // NL is newline token: '\r'? '\n'
row : STUFF+ ;
```

# Implementing an SDT in ANTLR4

- Enriched grammar with code

```
tour/Rows.g4
grammar Rows;

@parser::members { // add members to generated RowsParser
 int col;
 public RowsParser(TokenStream input, int col) { // custom constructor
 this(input);
 this.col = col;
 }
}

file: (row NL)+ ;

row
locals [int i=0]
: (STUFF
 {
 $i++;
 if ($i == col) System.out.println($STUFF.text);
 }
)+
;

TAB : '\t' -> skip ; // match but don't pass to the parser
NL : '\r'? '\n' ; // match and pass to the parser
STUFF: ~[\t\r\n]+ ; // match any chars except tab, newline
```

# Implementing an SDT in ANTLR4

- Running the parser (see code)

tour/Col.java

```
RowsLexer lexer = new RowsLexer(input);
CommonTokenStream tokens = new CommonTokenStream(lexer);
int col = Integer.valueOf(args[0]);
RowsParser parser = new RowsParser(tokens, col); // pass column number!
parser.setBuildParseTree(false); // don't waste time bulding a tree
parser.file(); // parse
```