Theme 2

ANTLR4

Introduction, Structure, Application

Compiladores, 2º semestre 2022-2023

Miguel Oliveira e Silva, Artur Pereira
DETI, Universidade de Aveiro

Presentation

ANTLR4: presentation

- *ANother Tool for Language Recognition*
- `ANTLR` is a language processor generator that can be used to read, process, execute or translate languages.
- Developed by Terrence Parr:

1988: master's thesis (`YUCC`)
1990: `PCCTS` (`ANTLR v1`). Programmed in `C++`.
1992: `PCCTS v 1.06`
1994: `PCCTS v 1.21` and `SORCERER`
1997: `ANTLR v2`. Programmed in `Java`.
2007: `ANTLR v3` (`LL(*)`, *auto-backtracking*, yuk!).
2012: `ANTLR v4` (`ALL(*)`, *adaptive LL*, yep!).

- Terrence Parr, The Definitive ANTLR 4 Reference, 2012, The Pragmatic Programmers.
- Terrence Parr, Language Implementation Patterns, 2010, The Pragmatic Programmers.
- `https://www.antlr.org`

ANTLR4: installation

- Download the `antlr4-install.zip` file from *elearning*.
- Run the *script* `./install.sh` in the `antlr4-install` directory.
- There are two important `jar` files:

  `antlr-4.*-complete.jar` and `antlr-runtime-4.*.jar`
- The first is *required* for generating language processors, and the second is *enough* for them to execute.
- To try it out just:

  ```
  java -jar antlr-4.*-complete.jar
  ```

  or:

  ```
  java -cp .:antlr-4.*-complete.jar org.antlr.v4.Tool
  ```
- `ANTLR4` provides a very flexible testing tool (implemented with the `antlr4-test` script):

  ```
  java org.antlr.v4.gui.TestRig
  ```
- We can run a grammar on any input, and get the list of generated *tokens*, the syntax tree (in a `LISP` format), or graphically display the syntax tree.

ANTLR4: installation (2)

- In this course several commands are available (in `bash`) to simplify (even more) the generation language processors:

`antlr4`	compilation of ANTLR-v4 grammars
`antlr4-test`	grammar debugging
`antlr4-clean`	deletion of files generated by ANTLR-v4
`antlr4-main`	generation of `main` class for grammar
`antlr4-visitor`	generation of a *visitor* class for the grammar
`antlr4-listener`	generation of a *listener* class for the grammar
`antlr4-build`	compiles grammars and generated `java` code
`antlr4-run`	runs the class `*Main` associated with the grammar
`antlr4-jar-run`	run a jar file (including antlr jars)

ANTLR4: installation (3)

`antlr4-javac`	java compiler (antlr jar in CLASSPATH)
`antlr4-java`	java virtual machine (antlr jar in CLASSPATH)
`java-clean`	delete binary files `java`
`view-javadoc`	opens documentation for a `java` class in *browser*
`st-groupfile2string`	converts a `STGroupFile` to a `STGroupString`

- These commands are available in *elearning* and are part of the automatic installation.

Examples

ANTLR4: *Hello*

- ANTLR4:

text → LEXER → tokens → PARSER → parse-tree

- Example:

```
//  (this is a line comment)
grammar Hello;                  // Define a grammar called Hello
// parser (first letter in lower case):
r : 'hello' ID ;                // match keyword hello followed by an identifier
// lexer (first letter in upper case):
ID : [a-z]+ ;                   // match lower-case identifiers
WS : [ \t\r\n]+ -> skip ;       // skip spaces, tabs, newlines, (Windows)
```

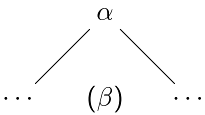
- The two grammars – lexical and syntactic – are expressed with instructions with the following structure:

$$\alpha : \beta;$$

where α corresponds to a single lexical or syntactic symbol (depending on your first letter be, respectively, uppercase or lowercase); and where β is a symbolic expression equivalent to α.

ANTLR4: *Hello* (2)

- A sequence of symbols in the input that is recognized by this grammar rule can always be expressed by a tree-like structure (called *syntactic*), where the root corresponds to α and the branches to sequence of symbols expressed in β:



- We can now generate the processor for this language and try out the grammar using the `ANTLR4` test program.

```
antlr4 Hello.g4
antlr4-javac Hello*.java
echo "hello compilers" | antlr4-test Hello r -tokens
```

- Usage:

```
antlr4-test [<Grammar> <rule>] [-tokens | -tree | -gui]
```

ANTLR4: Generated files

- Executing the command `antlr4` on this grammar we obtain the following files:

$$
\texttt{Hello.g4} \xrightarrow{\texttt{antlr4}}
\begin{array}{l}
\left.\begin{array}{l}\texttt{HelloLexer.java}\\ \texttt{HelloLexer.tokens}\\ \texttt{Hello.tokens}\end{array}\right\}\ \text{lexer}\\
\left.\begin{array}{l}\texttt{HelloParser.java}\end{array}\right\}\ \text{parser}\\
\left.\begin{array}{l}\texttt{HelloListener.java}\\ \texttt{HelloBaseListener.java}\end{array}\right\}\ \begin{array}{l}\text{syntax-tree}\\ \text{traversal}\end{array}
\end{array}
$$

ANTLR4: Generated files (2)

- Generated files:
  - `HelloLexer.java`: `Java` code with lexical analysis (generates *tokens* for parsing)
  - `Hello.tokens` and `HelloLexer.tokens`: files with the identification of *tokens* (not important at this stage, but it serves to modularize different lexical analyzers and/or to separate the lexical analysis from the analysis syntactic)
  - `HelloParser.java`: `Java` code with parsing (generates the parsing tree of the program)
  - `HelloListener.java` and `HelloBaseListener.java`: `Java` code that automatically implements a code execution pattern like *listener* (*observer*, *callbacks*) at all entry and exit points of all compiler syntactic rules.

ANTLR4: Generated files (2)

- We can run `ANTLR4` with the `-visitor` option to also generate `Java` code for the *visitor* type pattern (it differs from *listener* because the visit has to be explicitly required).
  - `HelloVisitor.java` and `HelloBaseVisitor.java`: `Java` code that automatically implements a *visitor* code execution pattern for all entry and exit points for all rules compiler syntax.

ANTLR4: `Expr`

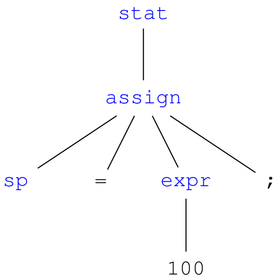
- Example:

```
grammar Expr;
stat: assign ;
assign: ID '=' expr ';' ;
expr: INT ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

- If we run the compiler created with the input:

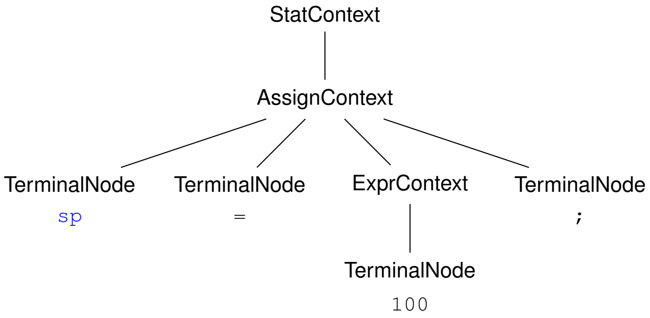
```
sp = 100;
```

- Let's get the following syntax tree:



ANTLR4: automatic context

- To facilitate semantic analysis and synthesis, `ANTLR4` tries to help with automatic resolution of many problems (as is the case with *visitors* and *listeners*)
- In the same sense, classes (and the respective objects in execution) are generated with the context of all grammar rules:



ANTLR4: automatic context (2)

```
grammar Expr;

stat : assign ;

assign : ID '=' expr ';' ;

expr : INT ;

ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

classes: ExprLexer and ExprParser

class StatContext in ExprParser

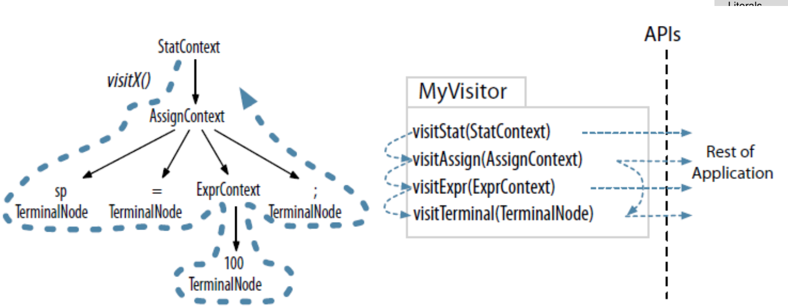
class AssignContext in ExprParser

class ExprContext in ExprParser

```
public class ExprParser extends Parser {

  public static class StatContext extends ParserRuleContext {

    public AssignContext assign() {
      ...
    }
    ...
  }
  ...
}
```

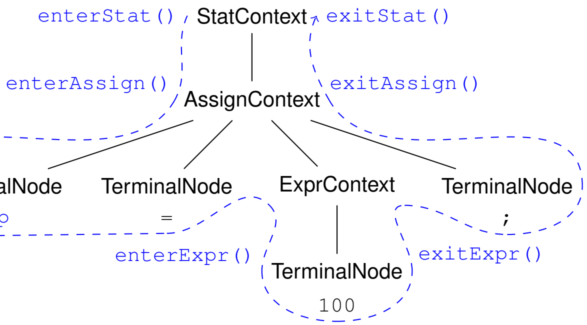
ANTLR4: *visitor*

- Context objects have all relevant parsing information associated with them (*tokens*, references to tree child nodes, etc.)
- For example the context `AssignContext` contains methods `ID` and `expr` to access the respective nodes.
- In the case of automatically generated code of the *visitor* type, the invocation pattern is illustrated below:



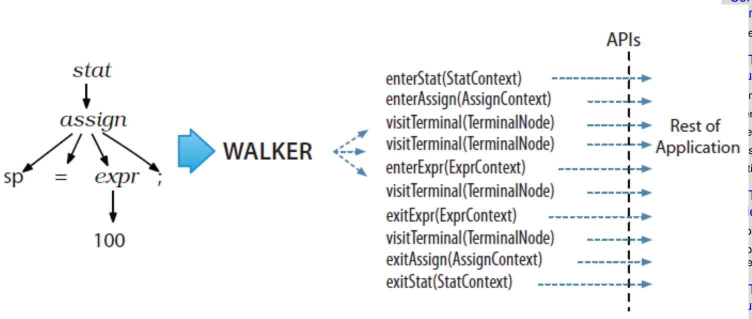
ANTLR4: *listener*

- The automatically generated code of type *listener* has the following invocation pattern:



ANTLR4: *listener* (2)

- Its connection to the rest of the application is as follows:



ANTLR4: attributes and actions

- It is possible to associate attributes and actions to rules:

```
grammar ExprAttr;
stat: assign ;
assign: ID '=' e=expr ';'
   {System.out.println($ID.text+" = "+$e.v);} // action
   ;
expr returns[int v]: INT  // result attribute named v in expr
   {$v = Integer.parseInt($INT.text);}        // action
   ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

- Unlike *visitors* and *listeners*, actions are executed during the syntactic analysis.
- The execution of each action takes place in the context where it is declared. So if an action is at the end of a rule (as exemplified above), its execution will occur after the respective recognition.
- The language to be executed in the action does not necessarily have to be Java (there are many others possible, such as C++ and python).

ANTLR4: attributes and actions (2)

- We can also pass attributes to the rule (like passing arguments to a method):

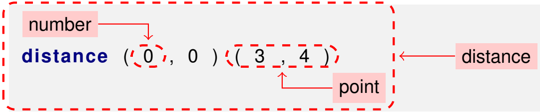
```
assign: ID '=' e=expr[true] ';' // argument passing to expr
   {System.out.println($ID.text+" = "+$e.v);}
   ;
expr[boolean a]      // argument attribute named a in expr
   returns[int v]:   // result attribute named v in expr
   INT {
      if ($a)
         System.out.println("Wow! Used in an assignment!");
      $v = Integer.parseInt($INT.text);
   } ;
```

- The similarity with passing method arguments and results is clear.
- Says attributes are synthesized when information comes from sub-rules, and inherited when information is sent to sub-rules.

Example Figures

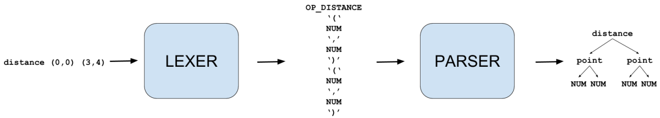
ANTLR4: Figures

- Retrieving the example from the figures.



- Initial grammar for figures:

```
grammar Shapes;
// parser rules:
distance: 'distance' point point;
point: '(' x=NUM ',' y=NUM ')';
// lexer rules:
NUM: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```



Integration in a program

```java
import java.io.IOException;
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;
public class ShapesMain {
    public static void main(String[] args) {
        try {
            // create a CharStream that reads from standard input:
            CharStream input = CharStreams.fromStream(System.in);
            // create a lexer that feeds off of input CharStream:
            ShapesLexer lexer = new ShapesLexer(input);
            // create a buffer of tokens pulled from the lexer:
            CommonTokenStream tokens = new CommonTokenStream(lexer);
            // create a parser that feeds off the tokens buffer:
            ShapesParser parser = new ShapesParser(tokens);
            // begin parsing at distance rule:
            ParseTree tree = parser.distance();
            if (parser.getNumberOfSyntaxErrors() == 0) {
                // print LISP-style tree:
                // System.out.println(tree.toStringTree(parser));
            }
        }
        catch(IOException e) {
            e.printStackTrace();
            System.exit(1);
        }
        catch(RecognitionException e) {
            e.printStackTrace();
            System.exit(1);
        }
    }
}
```

Example *visitor*

- A first (clean) version of a *visitor* can be generated with the script `antlr4-visitor`
- Then we can change it, for example, as follows:

```java
import org.antlr.v4.runtime.tree.AbstractParseTreeVisitor;

public class ShapesMyVisitor extends ShapesBaseVisitor<Object> {
  @Override
  public Object visitDistance(ShapesParser.DistanceContext ctx) {
    double res;
    double[] p1 = (double[])visit(ctx.point(0));
    double[] p2 = (double[])visit(ctx.point(1));
    res = Math.sqrt(Math.pow(p1[0]-p2[0],2) +
                    Math.pow(p1[1]-p2[1],2));
    System.out.println("visitDistance: "+res);
    return res;
  }

  @Override
  public Object visitPoint(ShapesParser.PointContext ctx) {
    double[] res = new double[2];
    res[0] = Double.parseDouble(ctx.x.getText());
    res[1] = Double.parseDouble(ctx.y.getText());

    return (Object)res;
  }
}
```

Example *visitor* (2)

- To use this class:

```
public static void main(String[] args) {
   ...
     // visitor:
   ShapesMyVisitor visitor = new ShapesMyVisitor();
   System.out.println("distance: "+visitor.visit(tree));
   ...
}
```

- The command `antlr4-main` allows the automatic generation of this code in the `main` method.

```
antlr4-main <Grammar> <start-rule>
            -v <visitor-class-or-file-name> ...
```

- Note that we can create the method `main` with as many *listeners* and *visitors* as we want (the order specified in the command arguments is maintained).

Example *listener*

```
import static java.lang.System.*;

import org.antlr.v4.runtime.ParserRuleContext;
import org.antlr.v4.runtime.tree.ErrorNode;
import org.antlr.v4.runtime.tree.TerminalNode;

public class ShapesMyListener extends ShapesBaseListener {
  @Override
  public void enterPoint(ShapesParser.PointContext ctx) {
    int x = Integer.parseInt(ctx.x.getText());
    int y = Integer.parseInt(ctx.y.getText());
    out.println("enterPoint x="+x+",y="+y);
  }

  @Override
  public void exitPoint(ShapesParser.PointContext ctx) {
    int x = Integer.parseInt(ctx.x.getText());
    int y = Integer.parseInt(ctx.y.getText());
    out.println("exitPoint x="+x+",y="+y);
  }
}
```

Example *listener* (2)

- To use this class:

```java
public static void main(String[] args) {
   ...
     // listener:
   ParseTreeWalker walker = new ParseTreeWalker();
   ShapesMyListener listener = new ShapesMyListener();
   walker.walk(listener, tree);
   ...
}
```

- The command `antlr4-main` allows the automatic generation of this code in the `main` method.

```
antlr4-main <Grammar> <start-rule>
            -l <listener-class-or-file-name> ...
```

Construction of grammars

Grammar Construction

- The construction of grammars can be considered a form of *symbolic programming*, in which there are symbols that are equivalent to sequences (that make sense) of other symbols (or even their own).
- The symbols used are divided into terminal and non-terminal symbols.
- Terminal symbols correspond to characters in lexical grammar and tokens in syntax; and non-terminal symbols are defined by productions (rules).
- In the end, all non-terminal symbols must be able to be expressed in terminal symbols.
- A grammar is constructed by specifying the rules or productions of grammatical elements.

```
grammar SetLang;      // a grammar example
stat: set set;        // stat is a sequence of two set
set: '{' elem* '}';   // set is zero or more elem inside { }
elem: ID | NUM;       // elem is an ID or a NUM
ID: [a-z]+;           // ID is a non-empty sequence of letters
NUM: [0-9]+;          // NUM is a non-empty sequence of digits
```

- Since its construction is a form of programming, we can benefit from identification and reuse of common problem solving patterns.

Construction of grammars (2)

- Surprisingly, the number of base patterns is relatively low:
  1. Sequence: sequence of elements;
  2. Optional: optional application of the element (zero or one occurrence);
  3. Repetitive: repeated application of the element (zero or more, one or more);
  4. Alternative: choose between different alternatives (for example, different types of instructions);
  5. Recursion: directly or indirectly recursive definition of an element (for example, conditional instruction is an instruction that selects other instructions for execution);
- Note that recursion and iteration are alternatives to each other. Assuming the existence of the empty sequence, the optional and repetitive patterns are implementable with recursion.
- However, as with programming in general, sometimes it is more appropriate to express recursion, and sometimes iteration.

Construction of grammars (3)

- Consider the following Java program:

```
import static java.lang.System.*;
public class PrimeList {
   public static void main(String[] args) {
      if (args.length != 1) {
         out.println("Usage: PrimeList <n>");
         exit(1);
      }
      int n = 0;
      try {
         n = Integer.parseInt(args[0]);
      }
      catch(NumberFormatException e) {
         out.println("ERROR: invalid argument '"+args[0]+"'");
         exit(1);
      }
      for(int i = 2;i <= n;i++)
         if(isPrime(i))
            out.println(i);
   }

   public static boolean isPrime(int n) {
      assert n > 1; // precondition

      boolean result = (n == 2 || n % 2 != 0);
      for(int i = 3;result && (i*i <= n);i+=2)
         result = (n % i != 0);
      return result;
   }
}
```

Construction of grammars (4)

- Even without an explicitly defined grammar, In this program, we can infer all the patterns mentioned above:
  1. String: The value assignment statement is defined as an identifier, followed by the character =, followed by an expression.
  2. Optional: the conditional statement may or may not have the code selection for the false condition.
  3. Repeating: (1) a class is a repetition of members; (2) an algorithm is a repetition of commands.
  4. Alternative: different instructions can be used where an instruction is expected.
  5. Recursion: the compound statement is defined as a sequence of statements delimited by braces; any of these statements can also be a compound statement.

Specification of grammars

- A language for specifying grammars needs to support this set of standards.
- To specify lexical elements (*tokens*) the notation used is based on *regular expressions*.
- The traditional notation used for parsing is called BNF (*Backus-Naur Form*).

  ```
  <symbol> ::= <meaning>
  ```
- This last notation originated in the construction of the language Algol (1960).
- ANTLR4 uses an altered and augmented variation (Extended BNF or EBNF) of this notation where you can define optional and repetitive constructions.

  ```
  <symbol> : <meaning> ;
  ```

ANTLR4: Lexical Structure

Lexical structure: comments

- The lexical structure of `ANTLR4` should be familiar to most programmers as it closely matches the syntax of languages in the `C` family (`C++`, `Java`, etc.).
- The comments are very similar to those of Java allowing the definition of line comments, multiline, or type `JavaDoc`.

```
/**
 * Javadoc alike comment!
 */
grammar Name;
/*
multiline comment
*/

/** parser rule for an identifier */
id: ID ; // match a variable name
```

Lexical structure: identifiers

- The first character of identifiers must be a letter, followed by other letters, digits or the character _
- If the first letter of the identifier is lower case, then this identifier represents a syntactic rule; otherwise (i.e. capital letter) then we are in the presence of a lexical rule.

```
ID, LPAREN, RIGHT_CURLY, Other // lexer token names
expr, conditionalStatment      // parser rule names
```

- As in `Java`, `Unicode` characters can be used.

Lexical structure: literals

- In ANTLR4 there is no distinction between character and *string* type literals.
- All literals are enclosed in single quotes.
- Examples: `'if'`, `'>='`, `'assert'`
- As in Java, literals can contain escape sequences like Unicode (`'\u3001'`), so like the usual escape sequences (`'\'\r\t\n'`)

Lexical structure: reserved words

- ANTLR4 has the following list of reserved words (i.e. that cannot be used as identifiers):

```
import, fragment, lexer,
parser, grammar, returns,
locals, throws, catch,
finally, mode, options,
tokens, skip
```

- Even though it is not a reserved word, you cannot use the word `rule` as that name conflicts with with the names generated in the code.

Lexical structure: actions

- Actions are blocks of code written in the target language (`Java` by default).
- Actions can have multiple locations within the grammar, but the syntax is always the same: brace-delimited text: `{...}`
- If by chance there are *strings* or comments (both like `C/Java`) containing curly braces there is no need to include an escape character (`{..."}"./*}*/..}`).
- The same happens if the braces were balanced (`{{...{}...}}`).
- Otherwise, you must use the escape character (`{\{}`, `{\}}`).
- Text included within actions must conform to the target language.

Lexical structure: actions (2)

- Actions can appear in lexical rules, in syntactic rules, in specifying exceptions of the grammar, in the attribute sections (result, argument and local variables), in certain sections of the header of the grammar and in some rule options (semantic predicates).
- Each action can be assumed to be executed in the context in which it appears (for example, at the end of recognition of a rule).

```
grammar Expr;
stat:
    {System.out.println("[stat]: before assign");} assign
  | expr {System.out.println("[stat]: after expr");}
  ;
assign:
  ID
  {System.out.println("[assign]: after ID and before =!");}
  '=' expr ';';
expr: INT {System.out.println("[expr]: INT!");} ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

ANTLR4: Lexical Rules

Lexical rules

- The lexical grammar is composed of rules (or productions), where each rule defines a *token*.
- Lexical rules must start with a capital letter, and can only be visible within the lexical analyzer:

```
INT : DIGIT+   ;           // visible in both parser and lexer
fragment DIGIT : [0-9];    // visible only in lexer
```

- As, sometimes, the same sequence of characters can be recognized by different rules (for example: identifiers and reserved words), ANTLR4 establishes criteria that allow to eliminate this ambiguity (and thus, recognize one, and only one, textittoken).

Lexical rules (2)

- These criteria are essentially two (in the following order):
  1. Recognizes *tokens* that consume as many characters as possible.
     For example, in a lexical recognizer for Java, the text `ifa` is recognized with a single *token* type identifier, and not as two *tokens* (reserved word `if` followed by the identifier `a`).
  2. Gives priority to the rules defined first.
     For example, in the following grammar:

     ```
     ID: [a-z]+;
     IF: 'if';
     ```

     the *token* `IF` will never be recognized!
- ANTLR4 also assumes that implicitly defined *tokens* in syntactic rules, they are defined *before* those explicitly defined by lexical rules.
- The specification of these rules uses regular expressions.

Regular expressions in ANTLR4

Syntax	*Description*
$R : \ldots ;$	*Define lexer rule R*
X	*Match lexer rule element X*
$'literal'$	*Match literal text*
$[char\text{-}set]$	*Match one of the chars in* $char\text{-}set$
$'x'..'y'$	*Match one of the chars in the interval*
$X\ Y\ \ldots\ Z$	*Match a sequence of rule lexer elements*
$(\ldots)$	*Lexer subrule*
$X?$	*Optionally match rule element X*
$X*$	*Match rule element X zero or more times*
$X+$	*Match rule element X one or more times*

Regular expressions in ANTLR4(2)

Syntax	*Description*
$\sim x$	*Match one of the chars NOT in the set defined by x*
.	*Match any char*
$X{*}?\ Y$	*Match X until Y appears (non-greedy match)*
$\{\ldots\}$	*Lexer action*
$\{p\}?$	*Evaluate semantic predicate p (if false, the rule is ignored)*
$x \mid \ldots \mid z$	*Multiple alternatives*

Typical lexical patterns

Token category	Possible implementation

Identifiers	`ID: LETTER (LETTER | DIGIT)*;`
	`fragment LETTER: 'a'..'z'|'A'..'Z'|'_';`
	`// same as: [a-zA-Z_]`
	`fragment DIGIT: '0'..'9';`
	`// same as: [0-9]`

Numbers	`INT: DIGIT+;` `FLOAT: DIGIT+ '.' DIGIT+ | '.' DIGIT+;`
Strings	`STRING: '"' (ESC | . )*? '"';` `fragment ESC: '\\"' | '\\\\' ;`
Comments	`LINE_COMMENT: '//' .*? '\n' -> skip;` `COMMENT: '/*' .*? '*/' -> skip;`
Whitespace	`WS: [ \t\n\r]+ -> skip;`

"Non-greedy" Lexical Operator

“Non-greedy” lexical operator

- By default, lexical analysis is “greedy”.
- That is, *tokens* are generated with the largest possible size.
- This characteristic is generally desired, but it can cause problems in some cases.
- For example, if we want to recognize a *string*:

```
STRING: '"' .* '"';
```

- (In the lexical analyzer the dot (`.`) recognizes any character except `EOF`.)
- This rule does not work, because once the first `"` character is recognized, the lexical analyzer will recognize all characters as belonging to the `STRING` up to the last `"` character.
- This problem is solved with the *non-greedy* operator:

```
STRING: '"' .*? '"'; // match all chars until a " appears!
```

ANTLR4: Syntactic Structure

ANTLR4: Syntactic structure

- The grammars in ANTLR4 have the following syntactic structure:

```
grammar Name;          // mandatory
options { ... }        // optional
import ... ;           // optional
tokens { ... }         // optional
@actionName { ... }    // optional
rule1 : ... ;          // parser and lexer rules
...
```

- The lexical and syntactic rules can appear mixed up and are distinguished by the first letter whether the rule name is lowercase (parser) or uppercase (lexical parser).
- As already mentioned, the order in which the lexical rules are defined is very important.

ANTLR4: Syntactic structure (2)

- It is possible to separate syntactic grammars from lexicons by preceding the reserved word `grammar` with the reserved words `parser` or `lexer`.

```
grammar parser NameParser;
...
```

```
lexer grammar NameLexer;
...
```

- The options section allows you to define some options for parsers (e.g. origin of *tokens*, and the target programming language).

```
options { tokenVocab=NameLexer; }
```

- Any option can be overridden by arguments in `ANTLR4` invocation.
- The **`import`** relates to grammar inheritance (which we will see later).

tokens section

- The tokens section allows you to associate identifiers with *tokens*.
- These identifiers must then be associated with lexical rules, which can be in the same grammar, in another grammar, or even be directly programmed.

```
tokens { «Token1», ..., «TokenN» }
```

- For example:

```
tokens { BEGIN, END, IF, ELSE, WHILE, DO }
```

- Note that it is not necessary to have this section when the tokens come from an antlr4 lexical grammar (the `options` section with the `tokenVocab` variable correctly defined is enough).

Actions in grammar preamble

- This section allows the definition of actions in the preamble of the grammar (as we have already seen, actions can also exist in other areas of the grammar).
- There are currently only two possible actions in this area (with `Java` as target language): `header` and `members`

```
grammar Count;
@header {
package foo;
}
@members {
int count = 0;
}
```

- The first injects code at the beginning of files, and the second allows you to add members to the classes of the parser and/or lexicon.
- Eventually we can restrict these actions either to the parser (`@parser::header`) or to the lexical analyzer (`@lexer::members`)

ANTLR4: Syntactic Rules

Rule construction: synthesis

Syntax	***Description***
$r : \ldots ;$	*Define rule r*
x	*Match rule element x*
$x\ y\ \ldots\ z$	*Match a sequence of rule elements*
$(\ldots)$	*Subrule*
$x?$	*Match rule element x*
$x*$	*Match rule element x zero or more times*
$x+$	*Match rule element x one or more times*
$x \mid \ldots \mid z$	*Multiple alternatives*

A rule element is a token (lexical, or terminal rule), a syntactical rule (non-terminal), or a subrule.

Syntactic rules: moving information

- In ANTLR4 each syntactic rule can be seen as a kind of method, with similar communication mechanisms: arguments and result, as well as local variables to the rule.
- We can also annotate rules with an alternative name:

```
expr: e1=expr '+' e2=expr
    | INT;
```

- We can also label with names different alternatives of a rule:

```
expr: expr '*' e2=expr # ExprMult
    | expr '+' e2=expr # ExprAdd
    | INT              # ExprInt
    ;
```

- ANTLR4 will generate context information for each name (including methods to use in the *listener* and/or in *visitors*).

Syntactic rules: moving information (2)

```
grammar Info;

@header {
import static java.lang.System.*;
}

main: seq1=seq[true] seq2=seq[false] {
        out.println("average(seq1): "+$seq1.average);
        out.println("average(seq2): "+$seq2.average);
    }
    ;

seq[boolean crash] returns[double average=0]
    locals[int sum=0, int count=0]:
    '(' ( INT {$sum+=$INT.int;$count++;} )* ')' {
        if ($count > 0)
            $average = (double)$sum/$count;
        else if ($crash) {
            err.println("ERROR: divide by zero!");
            exit(1);
        }
    }
   ;

INT: [0-9]+;
WS: [ \t\n\r]+ -> skip;
```

Typical syntactic patterns

Pattern name	*Possible implementation*

Sequence	`x y ... z`
	`'[' INT+ ']'`
	`'[' INT* ']'`

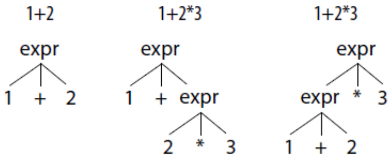
Sequence with terminator	`( instruction ';' )* // program sequence` `( row '\n' )* // lines of data`
Sequence with separator	`expr ( ',' expr)* // function call arguments` `( expr ( ',' expr)* )? // optional arguments`
Choice	`type: 'int' | 'float';` `instruction: conditional | loop | ... ;`

Token dependence	`'(' expr ')' // nested expression`
	`ID '[' expr ']' // array index`
	`'{' instruction+ '}' // compound instruction`
	`'<' ID (',' ID)* '>' // generic type specifier`

Recursivity	`expr: '(' expr ')' | ID;`
	`classDef: 'class' ID`
	`'{' (classDef|method|field)* '}';`

Precedence

- Sometimes, formally, the interpretation of the order of application of operators can be subjective:



- In ANTLR4 this ambiguity is resolved by giving priority to the declared subrules first:

```
expr: expr '*' expr // higher priority
    | expr '+' expr
    | INT           // lower priority
    ;
```

Associativity

- By default, associativity in applying the (same) operator is left to right:

  $a + b + c \;=\; ((a + b) + c)$

- However, there are operators, such as power, that may require inverse associativity:

  $a \uparrow b \uparrow c \;=\; a^{b^c} \;=\; a^{(b^c)}$

- This problem is solved in ANTLR4 as follows:

```
expr: <assoc=right> expr '^' expr
    | expr '*' expr // higher priority
    | expr '+' expr
    | INT            // lower priority
    ;
```

Grammar inheritance

- The import section implements an inheritance mechanism between grammars.
- For example the grammars:

```
grammar ELang;
stat : (expr ';')* EOF ;
expr : INT ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

```
grammar MyELang;
import ELang;
expr : INT | ID ;
ID : [a-z]+ ;
```

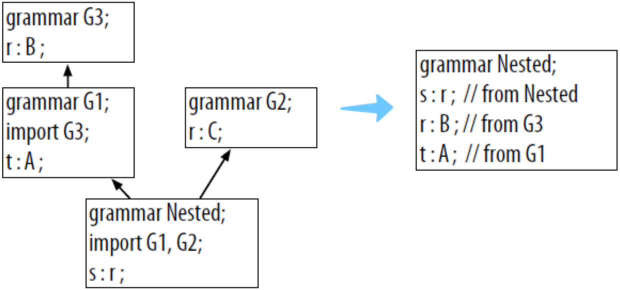
- Generate the equivalent `MyELang` grammar:

```
grammar MyELang;
stat : (expr ';')+ EOF ;
expr : INT | ID ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

- That is, rules are inherited, except when they are overridden in the descendant grammar.

Grammar inheritance (2)

- This mechanism allows multiple inheritance:



- Note the importance of the order of `imports` in the grammar Nested.
- The rule *r* comes from the grammar G3 and not from the grammar G2.

ANTLR4: More on actions

More on actions

- We have already seen that it is possible to add actions (expressed in the target language) directly to the grammar. that are executed during the parsing phase (in the order expressed in the grammar).
- We can also associate two special blocks of code to each rule – `@init` and `@after` – whose execution, respectively, precedes or follows the recognition of the rule.
- The `@init` block can be useful, for example, to initialize variables.
- The `@after` block is an alternative to placing the action at the end of the rule.

Example: CSV tables

Example

- Example: grammar for CSV files with the following requirements:
  1. The first line indicates the field names (must be written without any special formatting);
  2. In all lines other than the first, associate the value with the name of the field (they must be written with an explicit association, like assigning a value with `field = value`.

```
grammar CSV;

file: line line* EOF;

line: field (SEP field)* '\r'? '\n';

field: TEXT | STRING | ;

SEP: ','; // (' ' | '\t')*
STRING: [ \t]* '"' .*? '"' [ \t]*;
TEXT: ~[,"\r\n]~[,\r\n]*;
```

Example

```
grammar CSV;
@header {
import static java.lang.System.*;
}
@parser::members {
   protected String[] names = new String[0];
   public int dimNames() { ··· }
   public void addName(String name) { ··· }
   public String getName(int idx) { ··· }
}

file: line[true] line[false]* EOF;

line[boolean firstLine]
   locals[int col = 0]
   @after { if (!firstLine) out.println(); }
   : field[$firstLine,$col++] (SEP field[$firstLine,$col++])* '\r'? '\n';

field[boolean firstLine, int col]
   returns[String res = ""]
   @after {
      if ($firstLine)
         addName($res);
      else if ($col >= 0 && $col < dimNames())
         out.print("  "+getName($col)+": "+$res);
      else
         err.println("\nERROR: invalid field \""+$res+"\" in column "+($col+1));
   }
   :
   (TEXT {$res = $TEXT.text.trim();}) |
   (STRING {$res = $STRING.text.trim();}) |
   ;

SEP: ','; // (' ' | '\t')*
STRING: [ \t]* '"' .*? '"' [ \t]*;
TEXT: ~[,"\r\n]~[,\r\n]*;
```

ANTLR4: Ambiguous grammars

Ambiguous grammars

- The definition of grammars lends itself, with some ease, to generating ambiguities.
- This feature in human languages is sometimes sought after, but it's usually a problem.
  "To my advisor, to whom no thanks are too much."
  "The professor spoke to the engineering students"
  "What rimes with orange? . . . No it doesn't!"
- In the case of programming languages, where effects are to be interpreted and executed by machines (and not by us), there is no room for ambiguity.
- Thus, either by construction of the grammar or by priority rules that are applied to it by omission, grammars cannot be ambiguous.
- In ANTLR4 the definition and construction of rules defines priorities.

Ambiguous grammars: lexical analyzer

- If lexical grammars were only defined by regular expressions that compete with each other to consume input characters, then they would be naturally ambiguous.

```
...
conditional: 'if' '(' expr ')' 'then' stat; // incomplete
ID: [a–zA–Z]+;
...
```

- In this case the string **`if`** You can either give an identifier or a reserved word.
- ANTLR4 uses two rules outside regular expressions to handle ambiguity:
  1. By default, choose the *token* that consumes the maximum number of characters from the input;
  2. Gives priority to *tokens* defined first (whereas those defined implicitly in the grammar syntax take precedence over all others).

Ambiguous grammars: parser

- We have already seen that syntactic rules can also contain ambiguity.
- The following two excerpts exemplify ambiguous grammars:

```
stat: ID '=' expr
    | ID '=' expr
    ;
expr: NUM
    ;
```

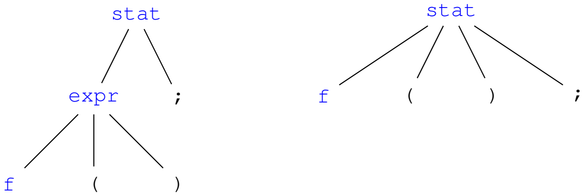
```
stat: expr ';'
    | ID '(' ')' ';'
    ;
expr: ID '(' ')'
    | NUM
    ;
```

- In both cases the ambiguity results from having a repeated subrule, directly in the first case and indirectly in the second case.

Ambiguous grammars: parser (2)

- The grammar is said to be ambiguous because, for the same input, we could have two trees different syntactics.

Expression `f();` Instruction `f();`



- Other examples of ambiguity are the precedence and associativity of operators.

Ambiguous grammars: parser (3)

- ANTLR4 has additional rules to eliminate syntactic ambiguities.
- As with the lexical analyzer, *Ad hoc* rules outside the context-independent grammar notation guarantee unambiguity.
- These rules are as follows:
  1. The alternatives, directly or indirectly, defined first have precedence over the rest.
  2. By default, operator associativity is left.
- Of the two parse trees presented in the previous example, the defined grammar imposes the first alternative.

Ambiguous grammars: parser (4)

- The C language has yet another practical example of ambiguity.
- The expression `i*j` either it can be a multiplication of two variables, as the declaration of a variable `j` as a pointer to the `i` data type.
- These two very different meanings can also be resolved in ANTLR4 grammars with the so-called semantic predicates.

ANTLR4: Semantic predicates

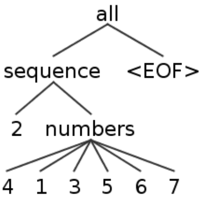
Semantic predicates

- In `ANTLR4` it is possible to use semantic information (expressed in the target language and injected into the grammar), to guide the parser.
- This functionality is called semantic predicates: `{...}?`
- Semantic predicates allow you to selectively enable/disable portions of grammar rules during parsing itself.
- Let's, as an example, develop a grammar to parse sequences of integers, but in which the first number does not belong to the sequence, but rather indicates the dimension of the sequence:
- So the list `2 4 1 3 5 6 7` would indicate two sequences: `(4,1) (5,6,7)`

Example

```
grammar Seq;

all: sequence* EOF;

sequence: INT numbers;

numbers: INT*;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

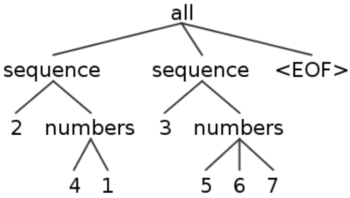
With this grammar, the parse tree generated for the input `2 4 1 3 5 6 7` is:



Example

```
grammar Seq;

all: sequence* EOF;

sequence
   @init { System.out.print("("); }
   @after { System.out.println(")"); }
   : INT numbers[$INT.int];

numbers[int count] locals [int c = 0]
   : ( {$c < $count}? INT
       {$c++; System.out.print(($c == 1 ? "" : " ")+$INT.text);}
     )* ;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Now the syntax tree already corresponds to what was intended:



ANTLR4: Separate *lexer* from *parser*

Separate lexical analyzer from parser

- Although it is possible to define the complete grammar, joining the lexical and syntactic analysis in the same module, we can also separate each of these grammars.
- This facilitates, for example, the reuse of lexical analyzers.
- There are also some features of the lexical analyzer, which require this separation (lexical “islands”).
- For the separation to be successful there are a set of rules that must be followed:
  1. Each grammar indicates its type in the header:
  2. Grammar names must (respectively) end in `Lexer` and `Parser`
  3. All *tokens* implicitly defined in the parser must be passed to the lexical parser (by assigning them an identifier for use in the *parser*).
  4. The lexical analyzer grammar must be compiled by `ANTLR4` before the parser grammar.
  5. The parser must include an option (`tokenVocab`) indicating the lexical analyzer.

Separate lexical analyzer from parser (2)

```
lexer grammar NAMELexer;
...
```

```
parser grammar NAMEParser;
options {
   tokenVocab=NAMELexer;
}
...
```

- In the grammar test, use the name without the suffix:

```
antlr4-test NAME rule
```

Example

```
lexer grammar CSVLexer;

COMMA: ',';
EOL: '\r'? '\n';
STRING: '"' ( '""' | ~'"' )* '"';
TEXT: ~[,"\r\n]~[,\r\n]*;
```

```
parser grammar CSVParser;

options {
   tokenVocab=CSVLexer;
}

file: firstRow row* EOF;

firstRow: row;

row: field (COMMA field)* EOL;

field: TEXT | STRING | ;
```

```
antlr4 CSVLexer.g4
antlr4 CSVParser.g4
antlr4-javac CSV*java
// or just: antlr4-build
antlr4-test CSV file
```

ANTLR4: Lexical "Islands"

“Lexical Islands”

- Another feature of `ANTLR4` is the possibility to recognize a different set of *tokens* according to certain criteria.
- For this purpose there are so-called lexical *modes*.
- For example, in `XML`, the lexical treatment of the text must be different depending on whether it is inside a *tag* or outside.
- A restriction of this feature is that you can only use lexical modes in lexical grammars.
- That is, the separation between the two types of grammars becomes mandatory.
- The lexical modes are managed by the commands: `mode(NAME)`, `pushMode(NAME)`, `popMode`
- The default lexical mode is called: `DEFAULT_MODE`

Example

```
lexer grammar ModesLexer;

// default mode

ACTION_START: '{' -> mode(INSIDE_ACTION);
OUTSIDE_TOKEN: ~'{'+;

mode INSIDE_ACTION;
ACTION_END: '}' -> mode(DEFAULT_MODE);
INSIDE_TOKEN: ~'}'+;
```

```
parser grammar ModesParser;

options {
   tokenVocab=ModesLexer;
}

all: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |
       INSIDE_TOKEN)* EOF;
```

Example (2)

```
lexer grammar ModesLexer;

// default mode

ACTION_START: '{' -> pushMode(INSIDE_ACTION);
OUTSIDE_TOKEN: ~'{'+;

mode INSIDE_ACTION;
ACTION_END: '}' -> popMode;
INSIDE_ACTION_START: '{' -> pushMode(INSIDE_ACTION);
INSIDE_TOKEN: ~[{}]+;
```

```
parser grammar ModesParser;

options {
   tokenVocab=ModesLexer;
}

all: ( ACTION_START | OUTSIDE_TOKEN | ACTION_END |
       INSIDE_ACTION_START | INSIDE_TOKEN)* EOF;
```

ANTLR4: Send *tokens* to different channels

Send *tokens* to different channels

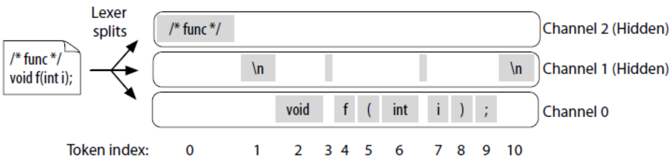
- In the examples of grammars that we have been presenting, the `skip` action has been chosen when in the presence of so-called blank spaces or comments.
- This action makes these *tokens* disappear, simplifying parsing.
- The price to pay (usually irrelevant) is losing the full text associated with them.
- However, in `ANTLR4` it is possible to have two in one. That is, to remove *tokens* from the syntactic analysis, without, however, making these *tokens* disappear completely (it is possible to recover the text associated with them).
- This is the role of the so-called lexical channels.

```
WS: [ \t\n\r]+           -> skip; // make token disappear
COMMENT: '/*' .*? '*/'   -> skip; // make token disappear
```

Send *tokens* to different channels (2)

```
WS: [ \t\n\r]+              -> channel(1); // redirect to channel 1
COMMENT: '/*' .*? '*/'  -> channel(2); // redirect to channel 2
```

- The `CommonTokenStream` class takes care of joining the tokens of all the channels (the visible one – channel zero – and the hidden ones).



- (It is possible to have code to access the *tokens* of a particular channel.)

Example: function declaration

```
grammar Func;

func: type=ID function=ID '(' varDecl* ')' ';';
varDecl: type=ID variable=ID;

ID: [a-zA-Z_]+;
WS: [ \t\r\n]+ -> channel(1);
COMMENT: '/*' .*? '*/' -> channel(2);
```

ANTLR4: Rewrite input

Rewrite input

- `ANTLR4` makes it easy to generate code that results from a rewrite of input code. That is, insert, delete, and/or modify parts of that code.
- For this purpose there is the `TokenStreamRewriter` class (which has methods for inserting text before or after *tokens*, or for erasing or replacing text).
- Let's assume that you want to make some changes to the `Java` source code, for example, add a comment immediately before the declaration of a class..
- We can fetch the available grammar for the 8 version of `Java`: `Java8.g4`

  (look at: `https://github.com/antlr/grammars-v4`)
- So that the rewrite only appends the comment, it is necessary to replace the `skip` of the *tokens* that are being ignored, redirecting them to a hidden channel.
- Now we can create a *listener* to solve this problem.

Example

```java
import org.antlr.v4.runtime.*;

public class AddClassCommentListener extends Java8BaseListener {

   protected TokenStreamRewriter rewriter;

   public AddClassCommentListener(TokenStream tokens) {
      rewriter = new TokenStreamRewriter(tokens);
   }

   public void print() {
      System.out.print(rewriter.getText());
   }

   @Override public void enterNormalClassDeclaration(
         Java8Parser.NormalClassDeclarationContext ctx) {
      rewriter.insertBefore(ctx.start, "/**\n * class "+
                                       ctx.Identifier().getText()+
                                       "\n */\n");
   }
}
```

ANTLR4:

Decouple code from grammar

ParseTreeProperty

Decouple code from grammar

- We have already seen that we can manipulate the information generated in parsing in multiple ways:
  - Directly in the grammar using actions and associating attributes with rules (arguments, result, local variables);
  - Using *listeners*;
  - Using *visitors*;
  - Associating attributes with the grammar by handling them within *listeners* and/or *visitors*.
- To associate extra information with the grammar, we can add attributes to the grammar (synthesized, inherited or local variables to the rules), or using the results of the `visit` methods.

Decouple grammar code (2)

- Alternatively, `ANTLR4` provides another possibility: its *runtime* library contains an associative *array* that allows you to associate parse tree nodes with attributes – `ParseTreeProperty`.
- Let's see an example with a grammar for arithmetic expressions:

Example

```
grammar Expr;

main: stat* EOF;

stat: expr;

expr: expr '*' expr # Mult
    | expr '+' expr # Add
    | INT           # Int
    ;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Example

```java
import org.antlr.v4.runtime.tree.ParseTreeProperty;

public class ExprSolver extends ExprBaseListener {
   ParseTreeProperty<Integer> mapVal = new ParseTreeProperty<>();
   ParseTreeProperty<String> mapTxt = new ParseTreeProperty<>();

   public void exitStat(ExprParser.StatContext ctx) {
      System.out.println(mapTxt.get(ctx.expr()) + " = " +
                         mapVal.get(ctx.expr()));
   }

   public void exitAdd(ExprParser.AddContext ctx) {
      int left = mapVal.get(ctx.expr(0));
      int right = mapVal.get(ctx.expr(1));
      mapVal.put(ctx, left + right);
      mapTxt.put(ctx, ctx.getText());
   }

   public void exitMult(ExprParser.MultContext ctx) {
      int left = mapVal.get(ctx.expr(0));
      int right = mapVal.get(ctx.expr(1));
      mapVal.put(ctx, left * right);
      mapTxt.put(ctx, ctx.getText());
   }

   public void exitInt(ExprParser.IntContext ctx) {
      int val  = Integer.parseInt(ctx.INT().getText());
      mapVal.put(ctx, val);
      mapTxt.put(ctx, ctx.getText());
   }
}
```