

# Tema 2

## ANTLR4

### Introdução, Estrutura, Aplicação

Linguagens Formais e Autómatos, 2<sup>o</sup> semestre 2017-2018

Miguel Oliveira e Silva, DETI, Universidade de Aveiro

### Conteúdo

<b>1</b>	<b>ANTLR: apresentação</b>	<b>3</b>
<b>2</b>	<b>Exemplos</b>	<b>3</b>
2.1	<i>Hello</i> . . . . .	4
2.2	<i>Expr</i> . . . . .	5
<b>3</b>	<b>Exemplo figuras</b>	<b>7</b>
3.1	Exemplo <i>listener</i> . . . . .	7
3.2	Exemplo <i>visitor</i> . . . . .	8
<b>4</b>	<b>Construção de gramáticas</b>	<b>8</b>
4.1	Especificação de gramáticas . . . . .	10
<b>5</b>	<b>ANTLR4: Estrutura sintáctica</b>	<b>10</b>
5.1	Secção de <i>tokens</i> . . . . .	10
5.2	Acções no preâmbulo da gramática . . . . .	11
<b>6</b>	<b>Estrutura léxica</b>	<b>11</b>
6.1	Comentários . . . . .	11
6.2	Identificadores . . . . .	11
6.3	Literais . . . . .	11
6.4	Palavras chave . . . . .	12
6.5	Acções . . . . .	12
<b>7</b>	<b>Regras léxicas</b>	<b>12</b>
7.1	Padrões léxicos típicos . . . . .	13
<b>8</b>	<b>Regras sintácticas</b>	<b>13</b>
8.1	Padrões sintácticos típicos . . . . .	15
8.2	Precedência . . . . .	15
8.3	Associatividade . . . . .	15
8.4	Herança de gramáticas . . . . .	16
<b>9</b>	<b>Operador léxico “não ganancioso”</b>	<b>16</b>
<b>10</b>	<b>Mais sobre acções</b>	<b>16</b>

<b>11 Gramáticas ambíguas</b>	<b>17</b>
<b>12 Predicados semânticos</b>	<b>19</b>
<b>13 Separar analisador léxico do analisador sintático</b>	<b>19</b>
<b>14 “Ilhas” lexicais</b>	<b>20</b>
<b>15 Enviar <i>tokens</i> para canais diferentes</b>	<b>21</b>
<b>16 Reescrever a entrada</b>	<b>22</b>
<b>17 Desacoplar código da gramática</b>	<b>23</b>

# 1 ANTLR: apresentação

- *ANother Tool for Language Recognition*
- O ANTLR é um gerador de processadores de linguagens que pode ser utilizado para ler, processar, executar ou traduzir linguagens.
- Desenvolvido por Terrence Parr:
  - 1988: tese de mestrado (YUCC)
  - 1990: PCCTS (ANTLR v1). Programado em C++.
  - 1992: PCCTS v 1.06
  - 1994: PCCTS v 1.21 e SORCERER
  - 1997: ANTLR v2. Programado em 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.

## ANTLR: instalação

- `http://www.antlr.org`
- Há dois ficheiros jar importantes:  
`antlr-4.7.1-complete.jar` e `antlr-runtime-4.7.1.jar`
- O primeiro é *necessário* para *gerar* processadores de linguagens, e o segundo é o *suficiente* para os *executar*.
- Para experimentar basta:  
`java -jar antlr-4.7.1-complete.jar`  
ou:  
`java -cp .:antlr-4.7.1-complete.jar org.antlr.v4.Tool`
- Pode copiar o primeiro ficheiro para uma pasta fixa: e.g. `/usr/java/packages/lib/ext/`
- O ANTLR fornece uma ferramenta de teste muito flexível:  
`java org.antlr.v4.gui.TestRig`
- Podemos executar uma gramática sobre uma qualquer entrada, e obter a lista de *tokens* gerados, a árvore sintáctica (num formato tipo LISP), ou mostrar graficamente a árvore sintáctica.
- Nesta disciplina são disponibilizados vários comandos (em `bash`) para simplificar (ainda mais) a geração de processadores de linguagens:

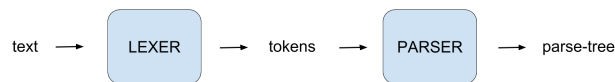
<code>antlr4</code>	compilação de gramáticas ANTLR-v4
<code>antlr4-test</code>	depuração de gramáticas
<code>antlr4-clean</code>	eliminação dos ficheiros gerados pelo ANTLR-v4
<code>antlr4-main</code>	geração da classe <code>main</code> para a gramática
<code>antlr4-build</code>	compila gramáticas e o código java gerado
<code>antlr4-run</code>	executa o compilador
<code>java-clean</code>	eliminação dos ficheiros binários java
<code>view-javadoc</code>	abre a documentação no <i>browser</i> de classes java
- Estes comandos estão disponíveis no elearning no ficheiro `antlr4-bin-v5.zip` (para documentação e instalação ler os ficheiros de texto lá existentes).

## 2 Exemplos

## 2.1 Hello

### ANTLR: Hello

- ANTLR:



- Exemplo:

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

- Podemos agora gerar o processador desta linguagem e experimentar a gramática utilizando a plataforma de teste do ANTLR.

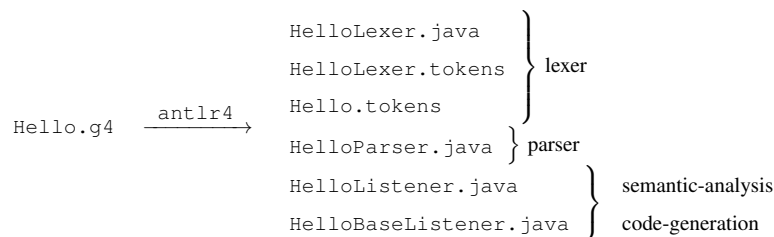
```
antlr4 Hello.g4
javac Hello*.java
echo "hello compiladores" | antlr4-test Hello r -tokens
```

- Utilização:

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

### ANTLR: Ficheiros gerados

- Executando o ANTLR sobre esta gramática obtemos os seguintes ficheiros:



- Ficheiros gerados:

- `HelloLexer.java`: código Java com a análise léxica (gera *tokens* para a análise sintática)
- `Hello.tokens` e `HelloLexer.tokens`: ficheiros com a identificação de *tokens* (pouco importante nesta fase, mas serve para modularizar diferentes analisadores léxicos e/ou separar a análise léxica da análise sintática)
- `HelloParser.java`: código Java com a análise sintática (gera a árvore sintática do programa)
- `HelloListener.java` e `HelloBaseListener.java`: código Java que implementa automaticamente um padrão de execução de código tipo *listener* (*callbacks*) em todos os pontos de entrada e saída de todas as regras sintáticas do compilador.

- Podemos executar o ANTLR com a opção `-visitor` para gerar também código Java para o padrão tipo *visitor* (difere do *listener* porque a visita tem de ser explicitamente requerida).

- `HelloVisitor.java` e `HelloBaseVisitor.java`: código Java que implementa automaticamente um padrão de execução de código tipo *visitor* todos os pontos de entrada e saída de todas as regras sintáticas do compilador.

## 2.2 Expr

### ANTLR: Expr

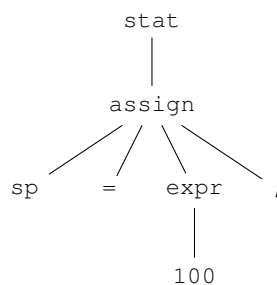
- Exemplo:

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

- Se executarmos o compilador criado com a entrada:

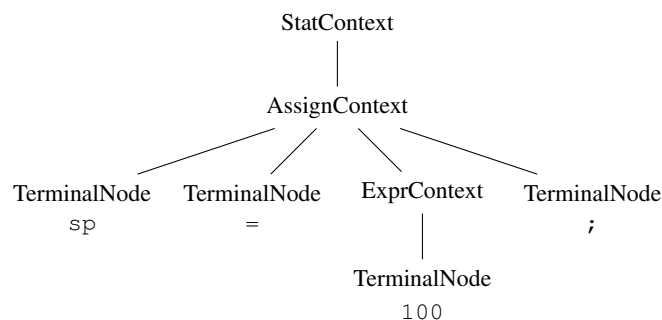
```
sp = 100;
```

- Vamos obter a seguinte árvore sintáctica:



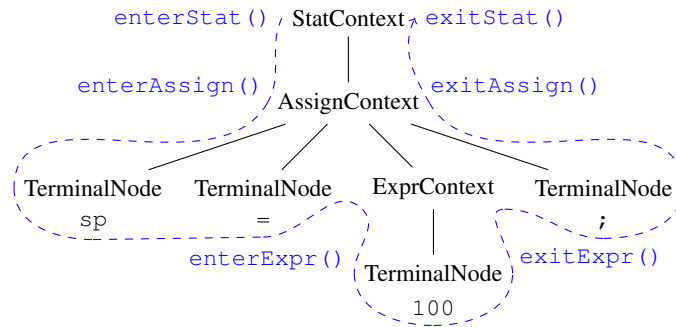
### ANTLR: contexto automático

- Para facilitar a análise semântica e a síntese, o ANTLR tenta ajudar na resolução automática de muitos problemas (como é o caso dos *listeners* e dos *visitors*)
- No mesmo sentido são geradas classes (e em execução os respectivos objectos) com o contexto de todas as regras da gramática:



### ANTLR: listener

- Os objectos de contexto têm a si associada toda a informação relevante da análise sintáctica (*tokens*, referência aos nós filhos da árvore, etc.)
- Por exemplo o contexto `AssignContext` contém métodos `ID` e `expr` para aceder aos respectivos nós.
- O código gerado automaticamente do tipo *listener* tem o seguinte padrão de invocação:



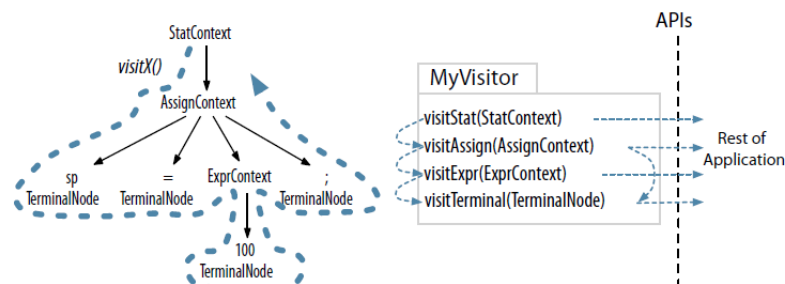
- A sua ligação à restante aplicação é a seguinte:



- (Já iremos ver como é que programaticamente se utiliza este código)

### ANTLR: *visitor*

- No caso do código gerado automaticamente do tipo *visitor* o padrão de invocação é ilustrado a seguir:



### ANTLR: atributos e acções

- É possível associar *atributos* e *acções* às regras:

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

- Também podemos passar atributos para a regra (tipo passagem de argumentos para um método):

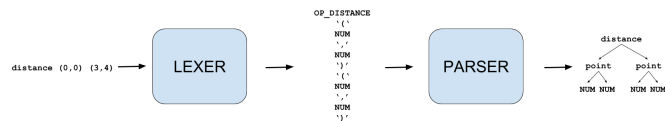
```
assign: ID '=' e=expr[true] ';' {
    System.out.println($ID.text+" = "+$e.v);
};
expr[boolean inAssign] returns [int v]: INT {
    if ($inAssign)
        System.out.println("Wow! Used in an assignment!");
    $v = Integer.parseInt($INT.text);
};
```

- É clara a semelhança com a passagem de argumentos e resultados de métodos.
- Diz que os atributos são *sintetizados* quando a informação provém de sub-regras, e *herdados* quando se envia informação para sub-regras.

### 3 Exemplo figuras

- Recuperando o exemplo das figuras da primeira aula.
- Gramática inicial para figuras:

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



#### Integração num programa

```
// import ANTLR's runtime libraries:
import org.antlr.v4.runtime.*;
import org.antlr.v4.runtime.tree.*;

public class ShapesMain {
    public static void main(String[] args) throws Exception {
        // create a CharStream that reads from standard input:
        ANTLRInputStream input = new ANTLRInputStream(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();
        // print LISP-style tree:
        System.out.println(tree.toStringTree(parser));
    }
}
```

- O comando `antlr4-main` permite a geração automática desta classe com uma primeira implementação do método `main`.

#### 3.1 Exemplo 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);
    }
}

```

- Para utilizar esta classe:

```

public static void main(String[] args) throws Exception {
    ...

    // listener:
    ParseTreeWalker walker = new ParseTreeWalker();
    ShapesMyListener listener = new ShapesMyListener();
    walker.walk(listener, tree);
}

```

- O comando `antlr4-main` permite a geração automática deste código no método `main`.  
`antlr4-main -l <nome-da-classe-ou-ficheiro-listener> ...`

### 3.2 Exemplo *visitor*

```

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;
    }
}

```

- Para utilizar esta classe:

```

public static void main(String[] args) throws Exception {
    ...

    // visitor:
    ShapesMyVisitor visitor = new ShapesMyVisitor();
    System.out.println("distance: "+visitor.visit(tree));
}

```

- O comando `antlr4-main` permite a geração automática deste código no método `main`.  
`antlr4-main -v <nome-da-classe-ou-ficheiro-visitor> ...`
- Note que podemos criar o método `main` com os *listeners* e *visitors* que quisermos (a ordem especificada nos argumentos do comando é mantida).

## 4 Construção de gramáticas

- As gramáticas são definidas recorrendo aos chamados *símbolos terminais e não terminais*, que são elementos gramaticais especificados por *tokens* ou por regras sintáticas mais abstractas.
- Uma gramática é construída especificando as *regras* ou construtores dos elementos gramaticais.



```

grammar SetLang;
stat: set set;
set: '{' elem* '}' ;
elem: ID | NUM;
ID: [a-z]+;
NUM: [0-9]+;

```

- A sua construção é uma forma de programação, e como tal, beneficia da identificação e reutilização de padrões comuns de resolução de problemas.
- Surpreendentemente, o número de padrões base é relativamente baixo:
  1. *Sequência*: sequência de elementos;
  2. *Optativo*: aplicação optativa do elemento (zero ou uma ocorrência);
  3. *Repetitivo*: aplicação repetida do elemento (zero ou muitas, uma ou muitas);
  4. *Alternativa*: escolha entre diferentes alternativas (como por exemplo, diferentes tipos de instruções);
  5. *Recursão*: definição directa ou indirectamente recursiva de um elemento (por exemplo, instrução condicional é uma instrução que selecciona para execução outras instruções);
- É de notar que a recursão e a iteração são alternativas entre si. Admitindo a existência da sequência vazia, os padrões optativo e repetitivo são implementáveis com recursão.
- No entanto, como em programação em geral, por vezes é mais adequado expressar recursão, e outras iteração.
- Considere o seguinte programa:

```

import static java.lang.System.*;
public class PrimeList {
    public static void main(String[] args) {
        if (args.length != 1) {
            out.println("Usage: PrimeList -ea <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;
    }
}

```

- Neste excerto de um programa Java, podemos inferir todos os padrões referidos na gramática da linguagem:
  1. *Sequência*: a instrução atribuição de valor é definida como sendo um identificador, seguido do carácter =, seguido de uma expressão.
  2. *Optativo*: a instrução condicional pode ter, ou não, a selecção de código para a condição falsa.
  3. *Repetitivo*: (1) uma classe é uma sequência de membros; (2) um algoritmo é uma sequência de comandos.

4. *Alternativa*: diferentes instruções podem ser utilizadas onde uma instrução é esperada.
5. *Recursão*: a instrução composta é definida como sendo uma sequência de instruções delimitada por chavetas; qualquer uma dessas instruções pode ser também uma instrução composta.

## 4.1 Especificação de gramáticas

- Uma linguagem para especificação de gramáticas precisa de suportar este conjunto de padrões.
- Para especificar elementos léxicos (*tokens*) a notação utilizada assenta em *expressões regulares*.
- A notação tradicionalmente utilizada para a análise sintáctica denomina-se por BNF (*Backus-Naur Form*).

```
<symbol> ::= <meaning>
```

- Esta última notação teve origem na construção da linguagem Algol (1960).
- O ANTLR utiliza uma variação alterada e aumentada (EBNF) desta notação onde se pode definir construções opcionais e repetitivas.

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

## 5 ANTLR4: Estrutura sintáctica

- As gramáticas em ANTLR têm a seguinte estrutura sintáctica:

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

- As regras léxicas e sintácticas pode aparecer misturadas e distinguem-se por a primeira letra do nome da regra ser minúscula (analizador sintáctico), ou maiúscula (analizador léxico).
- A ordem pela qual as regras léxicas são definidas é muito importante. Excepto no caso indicado a seguir, na presença duma ambiguidade, a primeira definição é a que conta.
- A excepção são os *tokens* literais definidos em regras sintácticas que têm precedência sobre os *tokens* definidos explicitamente por regras léxicas.
- É possível separar as gramáticas sintácticas das léxicas precedendo a palavra chave `grammar` com as palavras chave `parser` ou `lexer`.

```
parser grammar Name;
...
```

```
lexer grammar Name;
...
```

- A secção das *opções* permite definir algumas opções para os analisadores (e.g. origem dos *tokens*, e a linguagem de programação de destino).
- Qualquer opção pode ser redefinida por argumentos na invocação do ANTLR.
- A secção de `import` relaciona-se com herança de gramáticas (que veremos mais à frente).

### 5.1 Secção de *tokens*

- A secção de *tokens* permite associar identificadores a *tokens*.
- Esses identificadores devem depois ser associados a regras léxicas, que podem estar na mesma gramática, noutra gramática, ou mesmo ser directamente programados.

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

- Por exemplo: `tokens { BEGIN, END, IF, ELSE, WHILE, DO }`

## 5.2 Acções no preâmbulo da gramática

- Esta secção permite a definição de *acções* no preâmbulo da gramática (como já vimos, também podem existir acções noutras zonas da gramática).
- Actualmente só existem duas acções possíveis nesta zona (com o Java como linguagem destino): `header` e `members`

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

- A primeira injecta código no início de ficheiros, e a segunda permite que se acrescentem membros às classes do analisador sintáctico e/ou léxico.
- Eventualmente podemos restringir qualquer uma dessas acções ou ao analisador sintáctico (`@parser : header`) ou ao analisador léxico (`@lexer : members`)

## 6 Estrutura léxica

### 6.1 Comentários

- A estrutura léxica do ANTLR deverá ser familiar para a maioria dos programadores já que se aproxima da sintaxe das linguagens da família do C (C++, Java, etc.).
- Os comentários são em tudo semelhantes aos do Java permitindo a definição de comentários de linha, multilinha, ou tipo JavaDoc.

```
/**  
 * Javadoc alike comment!  
 */  
grammar Name;  
/*  
 multiline comment  
 */  
  
/** parser rule for an identifier */  
id: ID ; // match a variable name
```

### 6.2 Identificadores

- O primeiro carácter dos identificadores tem de ser uma letra, seguida por outras letras dígitos ou o carácter `_`
- Se a primeira letra do identificador é minúscula é uma regra sintáctica, se, por outro lado, for maiúscula estamos na presença duma regra léxica.

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

- Como em Java, podem ser utilizados caracteres Unicode.

### 6.3 Literais

- Em ANTLR não há distinção entre literais do tipo carácter e do tipo *string*.
- Todos os literais são delimitados por aspas simples.
- Exemplos: `'if'`, `'>='`, `'assert'`
- Como em Java, os literais podem conter sequências de escape tipo Unicode (`'\u3001'`), assim como as sequências de escape habituais (`'\r\t\n'`)

## 6.4 Palavras chave

- O ANTLR tem a seguinte lista de palavras reservadas (i.e. que não podem ser utilizadas como identificadores):

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

- Mesmo não sendo uma palavra reservada, não se pode utilizar a palavra `rule` já que esse nome entra em conflito com os nomes gerados no código.

## 6.5 Acções

- As acções são blocos de código escritos na linguagem destino (Java por omissão).
- As acções podem ter múltiplas localizações dentro da gramática, mas a sintaxe é sempre a mesma: texto arbitrário delimitado por chavetas: `{ ... }`
- Se por caso existirem *strings* ou comentários (ambos tipo C/Java) contendo chavetas não é necessidade de incluir um carácter de escape (`{ ... " " . / * } * / ... }`).
- O mesmo acontece se as chavetas foram balanceadas (`{ { ... { } ... } }`).
- Caso contrário, tem de se utilizar o carácter de escape (`{ \ { }, \ }`).
- O texto incluído dentro das acções tem de estar conforme com a linguagem destino.
- As acções podem aparecer nas regras léxicas, nas regras sintácticas, na especificação de excepções da gramática, nas secções de atributos (resultado, argumento e variáveis locais), em certas secções do cabeçalho da gramática e em algumas opções de regras (predicados semânticos).
- Pode considerar-se que cada acção será executada no contexto onde aparece (por exemplo, no fim do reconhecimento duma regra).

```
grammar Expr;
stat: assign ;
assign: ID '=' e=expr[true] ';' ;
    {System.out.println($ID.text+" = "+$e.v);} ;
expr[boolean inAssign] returns[int v]: INT {
    if ($inAssign)
        System.out.println("Used inside an assign!");
    $v = Integer.parseInt($INT.text);
} ;
ID : [a-z]+ ;
INT : [0-9]+ ;
WS : [ \t\r\n]+ -> skip ;
```

## 7 Regras léxicas

- A gramática léxica é composta por regras, que podem ser separadas em diferentes analisadores léxicos e compostas por diferentes modos (com regras léxicas distintas).
- As regras léxicas têm de começar por uma letra maiúscula, e podem ser visíveis apenas no analisador léxico:

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

- A especificação destas regras utiliza *expressões regulares*.

### Expressões regulares em ANTLR4

<b>Syntax</b>	<b>Description</b>
$R : \dots;$	Define lexer rule $r$
$X$	Match lexer rule element $X$
'literal'	Match literal text
[char-set]	Match one of the chars in char-set
'x'..'y'	Match one of the chars in the interval
$XY \dots Z$	Match a sequence of rule lexer elements
(...)	Lexer subrule
$X?$	Match rule element $X$
$X^*$	Match rule element $X$ zero or more times
$X^+$	Match rule element $X$ one or more times
$\sim x$	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)
{...}	Lexer action
{ $p$ }?	Evaluate semantic predicate $p$ (if false, the rule is ignored)
$x   \dots   z$	Multiple alternatives

## 7.1 Padrões léxicos típicos

<b>Token category</b>	<b>Possible implementation</b>
Identifiers	<pre>ID: LETTER (LETTER   DIGIT)*; <b>fragment</b> LETTER: 'a'..'z'   'A'..'Z'   '_' ; <b>fragment</b> DIGIT: '0'..'9' ;</pre>
Numbers	<pre>INT: DIGIT+; FLOAT: DIGIT+ '.' DIGIT+   '.' DIGIT+;</pre>
Strings	<pre>STRING: '"' (ESC   . ) *? '"'; <b>fragment</b> ESC: '\\"'   '\\\\' ;</pre>
Comments	<pre>LINE_COMMENT: '//' .*? '\n' -&gt; <b>skip</b>; COMMENT: '/*' .*? '*/' -&gt; <b>skip</b>;</pre>
Whitespace	<pre>WS: [ \t\n\r]+ -&gt; <b>skip</b>;</pre>

## 8 Regras sintáticas

Construção de regras: síntese

## **Syntax**   *Description*

*r* : ...;   *Define rule r*

*x*   *Match rule element x*

*xy...z*   *Match a sequence of rule elements*

(...)   *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*|...|*z*   *Multiple alternatives*

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

## Regras sintáticas: movendo informação

- Como já foi referido em ANTLR cada regras sintáctica é traduzida num método na linguagem destino (Java por omissão).
- Assim sendo é natural poder-se fazer uso dos mecanismos de comunicação entre métodos: *argumentos* e *resultado*, assim como poder-se definir *variáveis locais* à regra.
- Podemos também anotar regras com um nome alternativo:

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

- Podemos também dar nomes alternativos a diferentes alternativas duma regra:

```
expr: expr '*' e2=expr # Mult
    | expr '+' e2=expr # Add
    | INT;               # Int
```

- O ANTLR irá gerar informação de contexto para cada nome (incluindo métodos para usar no *listener* e/ou nos *visitors*).

**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]+;

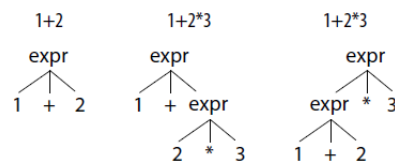
WS: [ \t\n\r]+ -> **skip**;

## 8.1 Padrões sintáticos típicos

Pattern name	Possible implementation
Sequence	<pre> x y ... z '[' INT+ ']' '[' INT* ']' </pre>
Sequence with terminator	<pre> ( instruction ';' ) * // program sequence ( row '\n' ) * // lines of data </pre>
Sequence with separator	<pre> expr ( ',' expr ) * // function call arguments ( expr ( ',' expr ) * ) ? // optional arguments </pre>
Choice	<pre> type: 'int'   'float'; instruction: conditional   loop   ... ; </pre>
Token dependence	<pre> '(' expr ')' // nested expression ID '[' expr ']' // array index '{' instruction+ '}' // compound instruction '&lt;' ID ( ',' ID ) * '&gt;' // generic type specifier </pre>
Nesting	<pre> expr: '(' expr ')'   ID; classDef: 'class' ID '{' (classDef/method/field) * '}' ; </pre>

## 8.2 Precedência

- A interpretação da ordem operadores é passível de subjectividade:



- Como já foi referido, em ANTLR esta ambiguidade é resolvida dando primazia às sub-regras declaradas primeiro:

```

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

```

## 8.3 Associatividade

- Por omissão, a associatividade na aplicação do (mesmo) operador é feita da esquerda para a direita:  
 $a + b + c = ((a + b) + c)$
- No entanto, há operadores, como é o caso da potência, que podem requerer a associatividade inversa:  
 $a \uparrow b \uparrow c = a^{b^c} = a^{(b^c)}$
- Este problema é resolvido em ANTLR de seguinte forma:

```

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

```

## 8.4 Herança de gramáticas

- A secção de *import* implementa um mecanismo de herança entre gramáticas.
- Por exemplo as gramáticas:

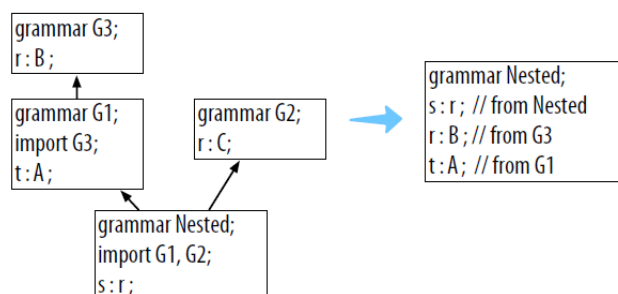
```
grammar ELang;  
stat : (expr ';'')+ ;  
expr : INT ;  
INT : [0-9]+ ;  
WS : [ \r\t\n]+ -> skip ;
```

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

- Geram a gramática MyELang equivalente:

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

- Isto é, as regras são herdadas, excepto quando são redefinidas na gramática descendente.
- Este mecanismo permite herança múltipla:



- Note-se a importância na ordem dos imports na gramática Nested.
- A regra *r* vem da gramática G3 e não da gramática G2.

## 9 Operador léxico “não ganancioso”

- Por omissão, a análise léxica é “gananciosa”.
- Isto é, os *tokens* são gerados com o maior tamanho possível.
- Esta particularidade é em geral a desejada, mas pode trazer problemas em alguns casos.
- Por exemplo, se quisermos reconhecer um *string*:

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

- (No analisador léxico o ponto (.) reconhece qualquer carácter excepto o EOF.)
- Esta regra não funciona, porque o analisador léxico vai reconhecer todos os caracteres como pertencendo ao STRING até ao EOF
- Este problema resolve-se com o operador *non-greedy*:

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

## 10 Mais sobre acções

- Já vimos que é possível acrescentar directamente na gramática acções (expressas na linguagem destino) que são executadas durante a fase de análise sintáctica (na ordem expressa na gramática).
- Podemos também associar a cada regra dois blocos especiais de código – @init e @after – cuja execução, respectivamente, precede ou sucede ao reconhecimento da regra.
- O bloco @init pode ser útil, por exemplo, para inicializar variáveis.
- O bloco @after é uma alternativa a colocar a acção no fim da regra.



- Exemplo: gramática para ficheiros tipo `csv` com os seguintes requisitos:
  1. A primeira linha indica o nome dos campos (deve ser escrita sem nenhuma formatação em especial);
  2. Em todas as linhas que não a primeira associar o valor ao nome do campo (devem ser escritas com a associação explícita, tipo atribuição de valor com `field = value`).

## Exemplo

```
grammar CSV;

file: line line* EOF;

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

field: TEXT | STRING | ;

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

## Exemplo

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

## 11 Gramáticas ambíguas

- A definição de gramáticas presta-se, com alguma facilidade, a gerar ambiguidades.
- Esta característica nas linguagens humanas é por vezes desejável (onde estaria a literatura e a poesia se não fosse assim) mas geralmente é um problema.

“Para o meu orientador, para quem nenhum agradecimento é demasiado.”

- No caso das linguagens de programação, em que os efeitos são para ser interpretados e executados por máquinas (e não por nós), não há espaço para ambiguidades.
- Assim, seja por construção da gramática, seja por regras de prioridade que lhe sejam aplicadas por omissão, as gramáticas não podem ser ambíguas.
- Em ANTLR4 a definição e construção de regras define prioridades.

### Gramáticas ambíguas: analisador léxico

- Se as gramáticas léxicas fossem apenas definidas por expressões regulares que competem entre si para consumir os caracteres de entrada, então elas seriam naturalmente ambíguas.

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

- Neste caso a sequência de caracteres **if** tanto pode dar um identificador como uma palavra reservada.
- O ANTLR4 utiliza duas regras fora das expressões regulares para lidar com ambiguidade:
  1. Por omissão, escolhe o *token* que consume o máximo número de caracteres da entrada;
  2. Dá prioridade aos *tokens* definidos primeiro (sendo que os definidos implicitamente na gramática sintáctica têm precedência sobre todos os outros).

### Gramáticas ambíguas: analisador sintáctico

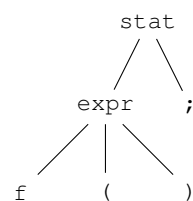
- Os dois excertos seguintes exemplificam gramáticas ambíguas:

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

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

- Em ambos os casos a ambiguidade resulta de ser ter uma sub-regra repetida, directamente, no primeiro caso, e indirectamente, no segundo caso.
- A gramática diz-se ambígua porque, para a mesma entrada, poderíamos ter duas árvores sintácticas diferentes.

Expressão `f () ;`



Instrução `f () ;`



- No entanto, em ANTLR4 isso não acontece. Isto é, tal como no analisador léxicos, as regras fora da notação das gramáticas livres de contexto, garantem a não ambiguidade.
- Em ANTLR4 a regra fora das gramáticas livres de contexto para lidar com ambiguidade é a seguinte:
  1. As alternativas, directa ou indirectamente, definidas primeiro têm precedência sobre as restantes.
- Das duas árvores sintácticas apresentadas no exemplo anterior, a gramática definida impõe a alternativa

- A linguagem C apresenta outros tipos de ambiguidade. A expressão  $i*j$  tanto pode ser uma multiplicação de duas variáveis, como a declaração de uma variável  $j$  como ponteiro para o tipo de dados  $i$ .
- Estes dois significados tão diferentes podem ser resolvidos em gramáticas ANTLR4 com os chamados *predicados semânticos*.

## 12 Predicados semânticos

- A ANTLR tem características que o tornam bem mais utilizável e programável que, por exemplo, os geradores de compiladores yacc/lex.
- Uma dessas características é a possibilidade de utilizar informação semântica (expressa na linguagem destino), para orientar o analisador sintático.
- Essa funcionalidade chama-se *predicados semânticos*:  $\{ \dots \}$ ?
- Os predicados semânticos permitem seletivamente activar/desactivar porções das regras gramaticais.
- Vamos, como exemplo, desenvolver uma gramática para analisar sequências de números inteiros, mas em que o primeiro número não pertence à sequência, mas indica sim a dimensão da sequência:
- Assim a lista 2 4 1 3 5 6 7 indicaria duas sequências: (4, 1) (5, 6, 7)

### Exemplo

```
grammar Seq;

all: sequence* EOF;

sequence: INT numbers;

numbers: INT+;

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

### Exemplo

```
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;
```

## 13 Separar analisador léxico do analisador sintático

- Muito embora se possa definir a gramática completa, juntando a análise léxica e a sintática no mesmo módulo, podemos também separar cada uma dessas gramáticas.
- Isso facilita, por exemplo, a reutilização de analisadores léxicos.

- Existem também algumas funcionalidades do analisador léxico, que obrigam a essa separação (“ilhas” lexicais).
- Para que a separação seja bem sucedida há um conjunto de regras que devem ser seguidas:
  1. Cada gramática indica o seu tipo no cabeçalho:
  2. Os nomes das gramáticas devem (respectivamente) terminar em `Lexer` e `Parser`
  3. Todos os *tokens* implicitamente definidos no analisador sintático têm de passar para o analisador léxico (associando-lhes um identificador para uso no *parser*).
  4. A gramática do analisador léxico deve ser compilada pelo ANTLR antes da gramática sintática.
  5. A gramática sintática tem de incluir uma opção (`tokenVocab`) a indicar o analisador léxico.

```
lexer grammar NAMELexer;
...

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

- No teste da gramática deve utilizar-se o nome sem o sufixo:

```
antlr4 -test NAME rule
```

## Exemplo

```
lexer grammar CSVLexer;

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

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
javac CSV*.java
// ou apenas: antlr4 -build
antlr4 -test CSV file
```

## 14 “Ilhas” lexicais

- Outra das características inovadoras do ANTLR é a possibilidade de reconhecer um conjunto diferente de *tokens* consoante determinados critérios.
- Para esse fim existem os chamados *modos* lexicais.
- Por exemplo, em XML, o tratamento léxico do texto deve ser diferente consoante se está dentro duma “marca” (*tag*) ou fora.

- Uma restrição desta funcionalidade é o facto de só se poderem utilizar modos lexicais em gramáticas léxicas.
- Ou seja, torna-se obrigatória a separação entre os dois tipos de gramáticas.
- Existem assim os comandos: `mode (NAME)`, `pushMode (NAME)`, `popMode`
- O modo lexical por omissão é designado por: `DEFAULT_MODE`

## Exemplo

```
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;

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;
```

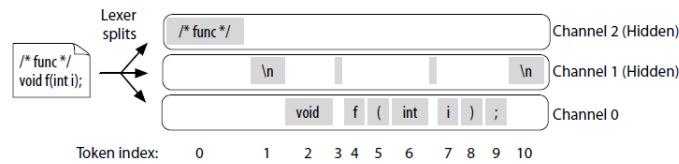
## 15 Enviar *tokens* para canais diferentes

- Nos exemplos de gramáticas que temos vindo a apresentar, tem-se optado pela acção `skip` quando na presença dos chamados espaços em branco ou de comentários.
- Esta acção faz desaparecer esses *tokens* simplificando a análise sintáctica.
- O preço a pagar (geralmente irrelevante) é perder o texto que lhes está associado.
- No entanto, em ANTLR é possível ter dois em um. Isto é, retirar *tokens* da análise sintáctica, sem no entanto fazer desaparecer completamente esses *tokens* (podendo-se recuperar o texto que lhe está associado).
- Esse é o papel dos chamados *canais léxicos*.

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

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

- A classe `CommonTokenStream` encarrega-se de juntar os tokens de todos os canais (o visível – canal zero – e os escondidos).



- (É possível ter código para aceder aos *tokens* de um canal em particular.)

## Exemplo: declaração de função

**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);

## 16 Reescrever a entrada

- O ANTLR facilita enormemente geração de código que é uma reescrita do código de entrada (com pequenas alterações).
- Para esse fim existe a classe `TokenStreamRewriter` (que têm métodos para inserir texto antes ou depois de *tokens*, ou para apagar ou substituir texto).
- Vamos supor que se pretende fazer ligeiras alterações no código fonte duma qualquer linguagem.
- Por exemplo, acrescentar um comentário imediatamente antes da declaração duma classe em Java.
- Podemos ir buscar a gramática disponível para a versão 8 do Java: `Java8.g4` (procurar em: <https://github.com/antlr/grammars-v4>)
- Para que a reescrita apenas acrescente o comentário, é necessário substituir o `skip` dos *tokens* que estão a ser desprezados, redireccionando-os para um canal escondido.
- Agora podemos criar um *listener* para resolver este problema.

### Exemplo

```
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");
    }
}
```

## 17 Desacoplar código da gramática

- Já vimos que podemos manipular a informação gerada na análise sintáctica de múltiplas formas:
  - Directamente na gramática recorrendo a acções e associando atributos a regras (argumentos, resultado, variáveis locais);
  - Utilizando *listeners*;
  - Utilizando *visitors*.
- No entanto, se quisermos separar completamente o código da gramática precisamos de soluções que não obriguem à colocação de código da linguagem destino na gramática.
- Das alternativas apresentadas, apenas a última (*visitors*) se aproxima desse requisito.
- No entanto, isso é feito com algumas restrições como é o caso do resultado do método `visit` ser sempre do mesmo tipo (pode-se utilizar coerções sobre o tipo `Object` mas essa prática deve ser evitada).
- No caso do *listener* o problema reside no desacoplamento entre os diferentes métodos o que torna a comunicação directa de informação mais difícil.
- Facilitamos essa comunicação se abdicarmos do requisito de não haver nenhum código da linguagem nativa na gramática e colocando lá resultados para as regras (que podem facilmente ser utilizados no contexto dos métodos *listener*).
- Alternativamente podemos simular a comunicação que existe entre métodos implementando explicitamente uma estrutura de dados tipo *stack* (mas isso é trabalhoso e sujeito a erros).
- Para desacoplamento total a melhor solução fornecida pelo ANTLR recorre a um mecanismo de biblioteca para anotar nós da árvore sintáctica com atributos.
- Esse suporte pela biblioteca recorre à classe `ParseTreeProperty`.
- Esta classe funciona como um *array* associativo em que a chave são nós da árvore sintáctica, e o elemento é do tipo que quisermos (até podemos ter mais do que um objecto `ParseTreeProperty` para diferentes tipos de elementos).
- Vamos ver um exemplo com uma gramática para expressões aritméticas:

### Exemplo

```
grammar Expr;

main: stat* EOF;

stat: expr;

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

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

### Exemplo

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

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());
    }
}

```