Tema 2

ANTLR4

Introdução, Estrutura, Aplicação

Compiladores, 2º semestre 2018-2019

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

Conteúdo

1 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.

ANTLR4: instalação

- http://www.antlr.org
- Há dois ficheiros jar importantes:
  antlr-4.7.2-complete.jar e antlr-runtime-4.7.2.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-4.7.2-complete.jar
```

  ou:

```
java -cp .:antlr-4.7.2-complete.jar org.antlr.v4.Tool
```

- Pode copiar o primeiro ficheiro para uma pasta fixa: e.g. /usr/java/packages/lib/ext/
- O ANTLR4 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:

antlr4	compilação de gramáticas ANTLR-v4
antlr4-test	depuração de gramáticas
antlr4-clean	eliminação dos ficheiros gerados pelo ANTLR-v4
antlr4-main	geração da classe main para a gramática
antlr4-build	compila gramáticas e o código java gerado
antlr4-run	executa o compilador
java-clean	eliminação dos ficheiros binários java
view-javadoc	abre a documentação no *browser* de classes java

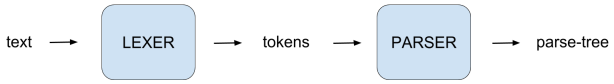
- Estes comandos estão disponíveis no elearning no ficheiro antlr4-bin-v5.2.zip (para documentação e instalação ler os ficheiros de texto lá existentes).

2 Exemplos

2.1 *Hello*

ANTLR4: *Hello*

- ANTLR4:



- Exemplo:

```
// (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)
```

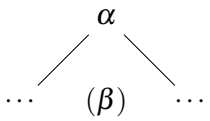
- As duas gramáticas – lexical e sintáctica – são expressas com instruções com a seguinte estrutura:

$$\alpha : \beta;$$

em que α corresponde a um único símbolo lexical ou sintáctico (dependendo da sua primeira letra ser, respectivamente, maiúscula ou minúscula); e em que β é uma expressão simbólica equivalente a α.

ANTLR4: *Hello* (2)

- Uma sequência de símbolos na entrada que seja reconhecido por esta regra gramatical pode sempre ser expressa por uma estrutura tipo árvore (chamada *sintáctica*), em que a raiz corresponde a α e os ramos à sequência de símbolos expressos em β:



- Podemos agora gerar o processador desta linguagem e experimentar a gramática utilizando o programa de teste do ANTLR4.

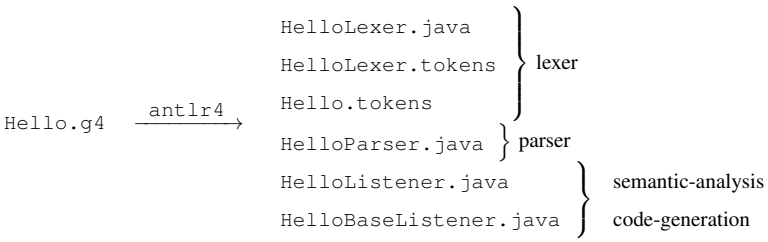
```
antlr4 Hello.g4

javac Hello*.java

echo "hello compiladores" | antlr4-test Hello r -tokens
```

- Utilização:

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

ANTLR4: Ficheiros gerados

- Executando o comando `antlr4` sobre esta gramática obtemos os seguintes ficheiros:



- Ficheiros gerados:
  - `HelloLexer.java`: código `Java` com a análise léxica (gera *tokens* para a análise sintáctica)
  - `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 sintáctica)
  - `HelloParser.java`: código `Java` com a análise sintáctica (gera a árvore sintáctica 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 sintácticas do compilador.

- Podemos executar o `ANTLR4` 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 sintácticas do compilador.

2.2 *Expr*

ANTLR4: `Expr`

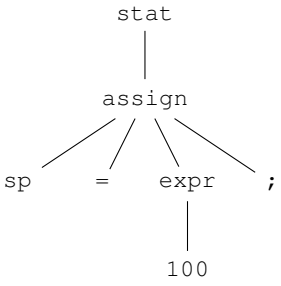
- Exemplo:

```
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:

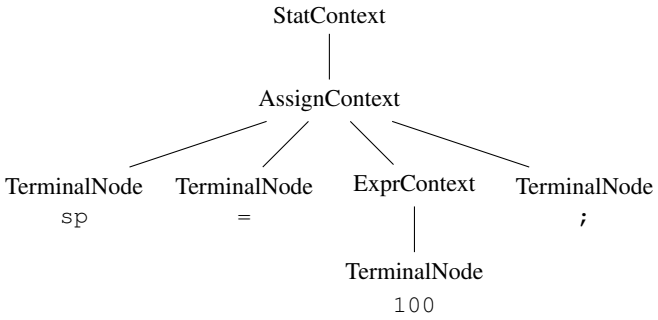
```
sp = 100;
```

- Vamos obter a seguinte árvore sintáctica:



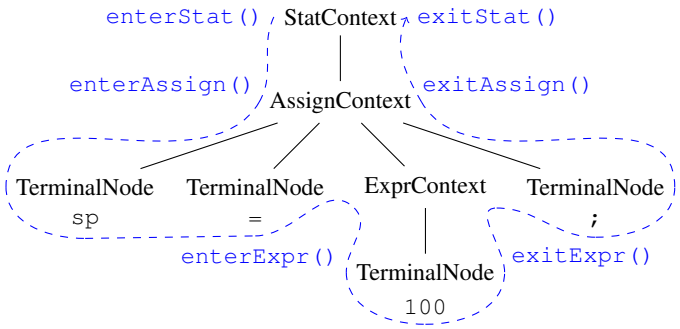
ANTLR4: contexto automático

- Para facilitar a análise semântica e a síntese, o `ANTLR4` 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:

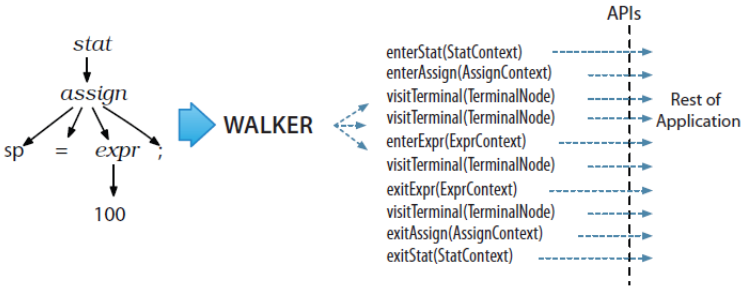


ANTLR4: *listener*

- 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:



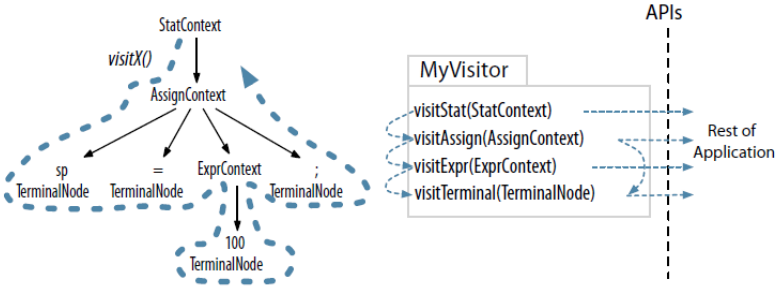
- A sua ligação à restante aplicação é a seguinte:



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

ANTLR4: *visitor*

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



ANTLR4: 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):

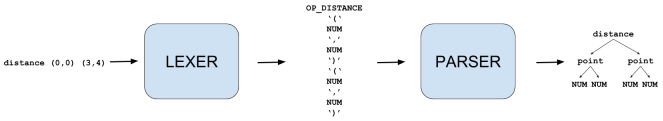
```
assign: ID '=' e=expr[true] ';'
   {System.out.println($ID.text+" = "+$e.v);} ;
expr[boolean inAssign] returns[int v]: INT {
      if ($inAssign)
         System.out.println("Wow! Used in an assignment!");
      $v = Integer.parseInt($INT.text);
   } ;
```

- 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.
- Gramática inicial para figuras:

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



Integração num programa

```
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:
      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);
      // replace error listener:
      //parser.removeErrorListeners(); // remove ConsoleErrorListener
      //parser.addErrorListener(new ErrorHandlingListener());
      // begin parsing at distance rule:
      ParseTree tree = parser.distance();
      if (parser.getNumberOfSyntaxErrors() == 0) {
         // print LISP-style tree:
         // System.out.println(tree.toStringTree(parser));
      }
```

```
    }
}
```

- O comando `antlr4-main` gera automaticamente esta 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 <Grammar> <start-rule> -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 <Grammar> <start-rule> -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

- A construção de gramáticas pode ser considerada uma forma de *programação simbólica*, em que existem símbolos que são equivalentes a sequências (que façam sentido) de outros símbolos (ou mesmo dos próprios).
- Os símbolos utilizados dividem-se em *símbolos terminais e não terminais*.
- Os símbolos terminais (ou *tokens*) são predefinidos, ou definidos fora da gramática; e os símbolos não terminais são definidos por produções (regras) da gramática (sendo estas transformações equivalentes de uma sequência de símbolos noutra sequência).
- No fim, todos os símbolos não terminais, com mais ou menos transformações, devem poder ser expressos em símbolos terminais.
- Uma gramática é construída especificando as *regras* ou produções dos elementos gramaticais.

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

- Sendo a sua construção uma forma de programação 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 mais, uma ou mais);
  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 em Java:

```
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;
    }
}
```

- Mesmo na ausência de uma gramática definida explicitamente, podemos neste programa inferir todos os padrões atrás referidos:
  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 `ANTLR4` utiliza uma variação alterada e aumentada (`EBNF`) desta notação onde se pode definir construções opcionais e repetitivas.

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

5 Estrutura sintáctica

- As gramáticas em `ANTLR4` 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 (analisador sintáctico), ou maiúscula (analisador 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 reservada `grammar` com as palavras reservadas `parser` ou `lexer`.

```
parser grammar NameParser;
...

lexer grammar NameLexer;
...
```

- A secção das *opções* permite definir algumas opções para os analisadores (e.g. origem dos *tokens*, e a linguagem de programação de destino).

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

- Qualquer opção pode ser redefinida por argumentos na invocação do `ANTLR4`.
- 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 }`
- Note que não é necessário ter esta secção quando os tokens tem origem numa gramática lexical antlr4 (basta a secção `options` com a variável `tokenVocab` correctamente definida).

5.2 Acções no preâmbulo da gramática

- Esta secção permite a definição de *acções* no preâmbulo da gramatica (como já vimos, também podem existem acções noutras zonas da gramática).
- Actualmente só existem dois 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 inicio de ficheiros, e a segunda permite que se acrescente membros às classes do analisador sintáctico e/ou léxico.
- Eventualmente podemos restringir estas acções ou ao analisador sintáctico (`@parser::header`) ou ao analisador léxico (`@lexer::members`)

6 Estrutura léxica

6.1 Comentários

- A estrutura léxica do `ANTLR4` 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 caráter dos identificadores tem de ser uma letra, seguida por outras letras dígitos ou o caráter _
- 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 caráteres `Unicode`.

6.3 Literais

- Em `ANTLR4` não há distinção entre literais do tipo carácter e do tipo *string*.
- Todos os literais são delimitador 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 reservadas

- O `ANTLR4` 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 há necessidade de incluir um caráter de escape (`{..."}"./*}*/..}`).
- O mesmo acontece se as chavetas foram balanceadas (`{{...{}...}}`).
- Caso contrário, tem de se utilizar o caráter 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

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-set*
$'x'..'y'$	*Match one of the chars in the interval*
$XY\ldots Z$	*Match a sequence of rule lexer elements*
$(\ldots)$	*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)*
$\{\ldots\}$	*Lexer action*
$\{p\}?$	*Evaluate semantic predicate p (if false, the rule is ignored)*
$x\|\ldots\|z$	*Multiple alternatives*

7.1 Padrões léxicos típicos

Token category	*Possible implementation*

Identifiers	`ID: LETTER (LETTER | DIGIT)*;`
	`fragment LETTER: 'a'..'z'|'A'..'Z'|'_';`
	`fragment DIGIT: '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;`

8 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 carácteres como pertencendo ao `STRING` até ao `EOF`
- Este problema resolve-se com o operador *non-greedy*:

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

9 Regras sintácticas

Construção de regras: síntese

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.

Regras sintácticas: movendo informação

- Como já foi referido em `ANTLR4` 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 `ANTLR4` 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;
```

9.1 Padrões sintácticos típicos

Pattern name	*Possible implementation*

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

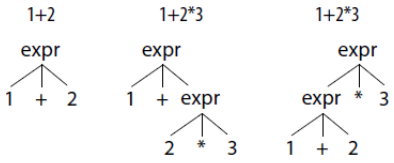
Sequence with terminator	`( instruction ';' )* // program sequence` `( row '\n' )*  // lines of data`
Sequence with separator	`expr ( ',' expr)*  // function call arguments` `( expr ( ',' expr)* )? // optional arguments`
Choice	`type: 'int'	'float';` `instruction: conditional	loop	... ;`

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

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

9.2 Precedência

- Por vezes, formalmente, a interpretação da ordem de aplicação de operadores pode ser subjectiva:



- Em ANTLR4 esta ambiguidade é resolvida dando primazia às sub-regras declaradas primeiro:

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

9.3 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 ANTLR4 de seguinte forma:

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

9.4 Herança de gramáticas

- A secção de *import* implementa um mecanismo de herança entre gramáticas.

- Por exemplo as gramáticas:

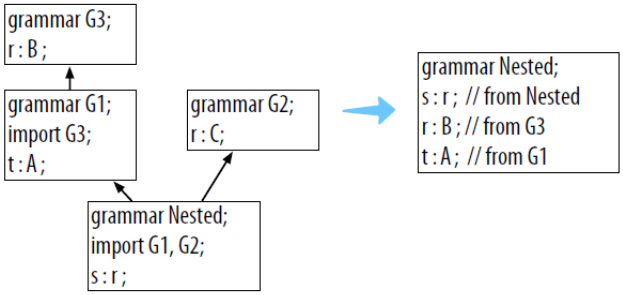
```
grammar ELang;
stat : (expr ';')* EOF ;
expr : INT ;
INT : [0-9]+ ;
WS : [ \r\t\n]+ -> skip ;
```

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

- Geram a gramática `MyELang` equivalente:

```
grammar MyELang;
stat : (expr ';')+ EOF ;
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.

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

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

11 Gramáticas ambíguas

- A definição de gramáticas presta-se, com alguma facilidade, a gerar ambiguidades.
- Esta característica nas linguagens humanas é por vezes procurada (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."
  >
  > "O professor falou aos alunos de engenharia"
  >
  > *"What rimes with orange? ... No it doesn't!"*

- 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

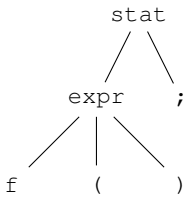
- Os dois excertos seguintes exemplificam gramáticas ambíguas:

```
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();`

- Outros exemplos de ambiguidade são os da precedência e associatividade de operadores (secções 9.2 e 9.3).
- O ANTLR4 tem regras adicionais para eliminar ambiguidades sintácticas.
- Tal como no analisador léxico, regras *Ad hoc* fora da notação das gramática independentes de contexto, garantem a não ambiguidade.
- Essas regras são as seguintes:
  1. As alternativas, directa ou indirectamente, definidas primeiro têm precedência sobre as restantes.
  2. Por omissão, a associatividade de operadores é à esquerda.
- Das duas árvores sintácticas apresentadas no exemplo anterior, a gramática definida impõe a primeira alternativa.
- A linguagem `C` tem ainda outro exemplo prático 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 também ser resolvidos em gramáticas ANTLR4 com os chamados *predicados semânticos*.

12 Predicados semânticos

- Em `ANTLR4` é possível utilizar informação semântica (expressa na linguagem destino e injetada na gramática), para orientar o analisador sintáctico.

- Essa funcionalidade chama-se *predicados semânticos*: `{...}?`
- Os predicados semânticos permitem seletivamente activar/desactivar porções das regras gramaticais durante a própria análise sintáctica.
- 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 sintáctico

- Muito embora se possa definir a gramática completa, juntando a análise léxica e a sintáctica 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 sintáctico 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 `ANTLR4` antes da gramática sintáctica.
  5. A gramática sintáctica 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';
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
javac CSV*java
// ou apenas: antlr4-build
antlr4-test CSV file
```

14 "Ilhas" lexicais

- Outra característica do `ANTLR4` 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: ~[{}]+;
```

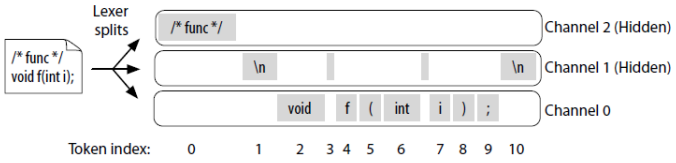
```
parser grammar ModesParser;

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 completo que lhes está associado.
- No entanto, em `ANTLR4` é possível ter dois em um. Isto é, retirar *tokens* da analise 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\n\r]+          -> 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 ANTLR4 facilita a geração de código que resulte de uma reescrita do código de entrada. Isto é, inserir, apagar, e/ou modificar partes desse código.
- 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 algumas alterações de código fonte Java, por exemplo, acrescentar um comentário imediatamente antes da declaração de uma classe..
- 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*;
  - Associando atributos à gramática fazendo a sua manipulação dentro dos *listeners* e/ou *visitors*.
- No entanto, se quisermos associar informação extra à gramática, até agora só o podíamos fazer acrescentando atributos à gramática (sintetizados, herdados ou variáveis locais às regras), ou utilizando os resultados dos métodos visit.
- A primeira destas opções, no entanto, representa uma dependência da gramática à linguagem destino escolhida.
- Uma possibilidade para resolver este problema consiste na simulação da comunicação existente entre métodos implementando explicitamente uma estrutura de dados tipo *stack* (mas isso é trabalhoso e sujeito a erros).
- O ANTLR4 fornece uma solução melhor: a sua biblioteca de *runtime* contém um *array* associativo que permite associar nós da árvore sintáctica com atributos – ParseTreeProperty.
- 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.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());
   }
}
```