# Block 2

# Introduction to ANTLR

**Summary:**

- Installation and use of ANTLR.
- Construction of simple grammars.
- Introduction to programming in ANTLR.

### Installation of ANTLR4

Download the file `antlr4-install.zip` available in elearning, extract its contents, open a terminal in the `antlr4-install` directory and finally run the command `./install.sh`.

### Exercise 2.01

Define the following grammar in the file `Hello.g4`[1]:

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

a) Try to compile the `Hello.g4` grammar, and run the resulting translator (use the command `antlr4-test` to test the grammar).

b) Add a *visitor* (`Execute`) so that, at the end of the rule `greetings`, write in Portuguese: `Olá <ID>`.

[1] Do not copy the code from the pdf file, as some copied characters may not be ASCII (generating errors when compiling with antlr4).

You can generate a new *visitor* in two ways:

(a) Run the following commands:

```
antlr4-visitor Hello.g4
mv HelloBaseVisitor.java Execute.java
```

, and then change the contents of the `Execute.java` file (change the class declaration so that `Execute` extend the `HelloBaseVisitor` class, and replace the generic type `T`, with the desired data type, which, in this case, is `String`).

(b) Or, alternatively, run the command:

```
antlr4-visitor Hello Execute String
```

Note that `ANTLR` for each syntactic rule `r: a b;`, creates a method `visitR` which has as argument the context of that rule (`*Parser.RContext ctx`). This context is defined by the `*Parser.RContext` class which, if `a` and `b` are non-terminal, contains methods that return the respective `*Parser.AContext` and `*Parser.BContext` contexts. If you want to visit one of these contexts, simply call, in the `visitR` method, for example, `visit(ctx.a())`.

c) Append a farewell rule to the grammar (`bye ID`), causing the grammar to accept either rule (`greetings`, or `bye`)[2]. (To define a rule with more than one alternative in `ANTLR`, use the separator `|`, for example: `r: a | b ;`.)

d) Generalize identifiers to include capital letters and allow names with more than one identifier. (In `ANTLR` a rule with one or more repetitions and defined by `r+`, for example: `r: a+ ;`.)

e) Generalize the grammar in order to allow repetition until the end of the rules file of any of the rules described above (`greetings`, or `bye`).

**Exercise 2.02**

Consider the following grammar (`SuffixCalculator.g4`):

```
grammar SuffixCalculator;
program:
    stat* EOF      // Zero or more repetitions of stat
  ;
stat:
    expr? NEWLINE  // Optative expr followed by an end-of-line
  ;
expr:
    expr expr op=('*'|'/'|'+'|'-')  #ExprSuffix
  | Number                          #ExprNumber
  ;
Number: [0-9]+('.'[0-9]+)?; // fixed point real number
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
```

[2]As a response to *bye*, write in Portuguese: `Adeus <ID>`.

a) Try compiling this grammar and running the resulting translator.

b) Using this grammar and adding a *visitor* (`Interpreter`), try to implement a calculator in postfix (suffix, or *Reverse Polish Notation*) notation for the defined elementary arithmetic operations (that is, that it performs the calculations and presents the results for each processed row).

   (Note that when implementing a *visitor* in ANTLR it is not necessary to apply the algorithm described in exercise 1.03 (which resorts to a *stack*).)

Note that in `ANTLR` we can associate *visits/callbacks* with different alternatives in a syntactic rule:

```
r : a #altA
  | b #altB
  | c #altC
  ;
```

In this case, *visits/callbacks* will be created both in *visitors* and in *listeners*, to the three alternatives presented, the *visit/callback* not appearing for the rule `r`.

**Exercise 2.03**

Consider the following grammar for an integer calculator (`Calculator.g4`):

```
grammar Calculator;

program:
      stat* EOF
    ;

stat:
      expr? NEWLINE
    ;

expr:
      expr op=('*'|'/'|'%') expr   #ExprMultDivMod
    | expr op=('+'|'-') expr       #ExprAddSub
    | Integer                      #ExprInteger
    | '(' expr ')'                 #ExprParent
    ;

Integer: [0-9]+;  // implement with long integers
NEWLINE: '\r'? '\n';
WS: [ \t]+ -> skip;
COMMENT: '#' .*? '\n' -> skip;
```

a) Try compiling this grammar and running the resulting translator.

b) Using this grammar and adding a *visitor* (`Interpreter`), try to implement a calculator for the defined elementary arithmetic operations (that is, to carry out the calculations and present the results to each line processed).

c) Add the unary operators + and −.

Note that the ambiguity between binary and unary operators that use the same symbol (+ and −) can only be resolved syntactically, and that these operators must take precedence over all others (for example, in the expression `-3+4` the unary operator − applies to `3` and not to the sum result).

**Exercise 2.04**
Taking into account the grammar of the 2.02 exercise, develop a prefix calculator. That is, where the binary operator appears **before** the two operands (as with the entry in the 1.07 exercise). Implement the interpreter with a *visitor*.

**Exercise 2.05**
Implement a grammar to parse the `numbers.txt` file used in the 1.04 exercise. Using a *listener*, change the part of the resolution of that problem that, in addition to reading that file, fills the associative *array*. The remaining part of the 1.04 problem should remain unchanged.

**Exercise 2.06**
Change problem 2.03 by adding the possibility to define and use variables. To that end consider a new statement (i.e. a new alternative in `stat`):

```
stat: ···  // make necessary changes
assignment: ID '=' expr;
···
expr:
   ···
   | ID                          #ExprId
   ···
ID: [a-zA-Z_]+ ;
···
```

To support recording values associated with variables, use an associative array[3].

**Exercise 2.07**
Download the `Java` language grammar (`https://github.com/antlr/grammars-v4`), and try parsing simple `Java` programs (version 8).

Use support for *listeners* from `ANTLR` to write the name of class and methods subject to parsing.

[3] java.util.HashMap.

**Exercise 2.08**
Using the grammar defined in the 2.06 problem and using *visitors* from `ANTLR`, convert an infix arithmetic expression (operator in the middle of the operands), into an equivalent suffix expression (operator at the end). To resolve the ambiguity problem with the unary operators + and −, make it so that in the converted suffix expression, these operators appear, respectively, as `!+` and `!-` . For example:

- $2 + 3 \quad \rightarrow \quad 2\ 3\ +$
- $a = 2 + 3 * 4 \quad \rightarrow \quad a\ =\ 2\ 3\ 4\ *\ +$
- $3 * (2 + 1) + (2 - 1) \quad \rightarrow \quad 3\ 2\ 1\ +\ *\ 2\ 1\ -\ +$
- $3 * +(4/2) + (-2 - 1) \quad \rightarrow \quad 3\ 4\ 2\ /\ \ !+\ *\ 2\ !-\ 1\ -\ +$

**Exercise 2.09**
We intend to implement a calculator for rational numbers (numerator and denominator represented by integers).

Define a grammar for this language taking into account the following example program (arithmetic operators must have the usual precedence):

```
// basic:
print 1/4;   // escreve na consola a fracção 1/4
print 3;     // escreve na consola a fracção 3
3/4 -> x;    // guarda a fracção 3/4 na variável x
print x;     // escreve na consola a fracção armazenada na variável x
// more advanced:
print 1/4-(1/4);
4/2 * (-2/3 + 4) - 2 -> x;
print x;
print x*x:x+(x)-x;   // a divisão de fracções é feita pelo operador :
print (1/2)^3;       // operador potência para expoentes inteiros (base entre parêntesis)
print reduce 2/4;    // redução de fracções
```

**Note 1**: Try to make the best possible use of the exemplified instructions in order to make the language as generic as possible. However, you can consider that literal fractions (e.g. 1/4, 2, −3/2) are always either an integer (i.e. unit denominator), or a ratio between two literal integers (where only the numerator can be negative).

**Note 2**: There are files `*.txt` that exemplify several programs.

**Note 3**: Starting in the program `p3.txt` there is an instruction `read` to allow the user to enter a fraction (i.e., at run time). Therefore, the source code must originate from a file instead of *standard input* (pass the file name as the program argument).

Implement this interpreter with *visitors*.