Lab 2
Lexing and Parsing with ANTLR4

Objective

- Understand the software architecture of ANTLR4.
- Be able to write simple grammars and correct grammar issues in ANTLR4.

EXERCISE #1 ▶ **Lab preparation**
In the `cap-labs` directory:

```
git pull
```

will provide you all the necessary files for this lab in `TP02`. You also have to install ANTLR4.

2.1 User install for ANTLR4 and ANTLR4 Python runtime

User installation steps:

```
mkdir ~/lib
cd ~/lib
wget http://www.antlr.org/download/antlr-4.7-complete.jar
pip3 install antlr4-python3-runtime --user
```

Then in your `.bashrc`:

```
export CLASSPATH=".:$HOME/lib/antlr-4.7-complete.jar:$CLASSPATH"
export ANTLR4="java -jar $HOME/lib/antlr-4.7-complete.jar"
alias antlr4="java -jar $HOME/lib/antlr-4.7-complete.jar"
alias grun='java org.antlr.v4.gui.TestRig'
```

Then source your `.bashrc`:

```
source ~/.bashrc
```

2.2 Structure of a .g4 file and compilation

Links to a bit of ANTLR4 syntax :

- Lexical rules (extended regular expressions): `https://github.com/antlr/antlr4/blob/4.7/doc/lexer-rules.md`
- Parser rules (grammars) `https://github.com/antlr/antlr4/blob/4.7/doc/parser-rules.md`

The compilation of a given `.g4` (for the PYTHON back-end) is done by the following command line:

```
java -jar ~/lib/antlr-4.7-complete.jar -Dlanguage=Python3  filename.g4
```

or if you modified your `.bashrc` properly:

```
antlr4 -Dlanguage=Python3  filename.g4
```

2.3 Simple examples with ANTLR4

EXERCISE #2 ▶ **Demo files**
Work your way through the five examples in the directory `demo_files`:

ex1 with ANTLR4 **+ Java** : A very simple lexical analysis[1] for simple arithmetic expressions of the form `x+3`. To compile, run:

```
antlr4 Example1.g4
javac *.java
```

This generates Java code then compile them and you can finally execute using the Java runtime with

```
grun Example1 tokens -tokens
```

To signal the program you have finished entering the input, use **Control-D** twice.
Examples of run: [`^D` means that I pressed Control-D]. What I typed is in boldface.

```
1+1
^D^D
[@0,0:0='1',<DIGIT>,1:0]
[@1,1:1='+',<OP>,1:1]
[@2,2:2='1',<DIGIT>,1:2]
[@3,4:3='<EOF>',<EOF>,2:0]
)+
^D^D
line 1:0 token recognition error at: ')'
[@0,1:1='+',<OP>,1:1]
[@1,3:2='<EOF>',<EOF>,2:0]
%
```

Questions:
- Read and understand the code.
- Allow for parentheses to appear in the expressions.
- What is an example of a recognized expression that looks odd? To fix this problem we need a syntactic analyzer (see later).

ex1bis : same with a PYTHON file driver:

```
antlr4 -Dlanguage=Python3 filename.g4
python3 main.py
```

Test the same expressions. Observe the PYTHON file.

From now on you can alternatively use the commands `make` and `make run` instead of calling `antlr4` and `python3`.

ex2 : Now we write a grammar for valid expressions. Observe how we recover information from the lexing phase (for ID, the associated text is `$ID.text$`).

If these files read like a novel, go on with the other exercises. Otherwise, do some testing and make sure that you understand what is going on. You can ask the Teaching Assistant, or another student, for advice.

EXERCISE #3 ▶ **Well-founded parenthesis**
Write a grammar and files to accept any text with well-formed parenthesis ')' and '['.

Important remark From now on, we will use Python at the right-hand side of the rules. As Python is sensitive to indentation, there might be some issues when writing on several lines. Try to be as compact as you can!

EXERCISE #4 ▶ **Towards analysis: If then else ambiguity** [2] **- Skip if you are late**
We give you the following grammar for imbricated "ifs" :

[1] Lexer Grammar in ANTLR4 jargon
[2] Also known as "dangling else"

```
grammar ITE;

prog: stmt;

stmt : ifStmt | ID ;

ifStmt : 'if' ID stmt ('else' stmt)? ;


ID : [a-zA-Z]+;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Find a way (with right actions) to test if:

```
if x if y a else b
```

is parsed as :

```
if x (if y a else b)
```

or

```
if x (if y a) else b
```

Thus ANTLR4 finds a way to decide which rule to "prioritize". There are other ways of getting around this problem explicitely:

- by changing the syntax: add explicit parentheses, or other block marks ("if .. then .. end else .. end")
- by disambiguating the grammar:
  ```
  ifStmt : 'if' ID ifStmt | 'if' ID ifThenStmt 'else' ifStmt
  ifThenStmt : 'if' ID ifThenStmt 'else' ifThenStmt
  ```
- by using the ANTLR4 "lookahead" concept.
  ```
  ifStmt : 'if' ID stmt ('else' stmt | {self.input.LA(1) != ELSE}?);
  ELSE : 'else';
  ```

more on `https://en.wikipedia.org/wiki/Dangling_else`

2.4 Grammar Attributes (actions)

Until now, our analyzers are passive oracles, ie language recognizers. Moving towards a "real compiler", a next step is to execute code during the analysis, using this code to produce an intermediate representation of the recognized program, typically ASTs. This representation can then be used to generate code or perform program analysis (see next labs). This is what *attribute grammars* are made for. We associate to each production a piece of code that will be executed each time the production will be reduced. This piece of code is called *semantic action* and computes attributes of non-terminals.

Let us consider the following grammar (the end of an expression is a semicolon):

$$
\begin{aligned}
Z &\rightarrow E; \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow id \\
F &\rightarrow int \\
F &\rightarrow (E)
\end{aligned}
$$

EXERCISE #5 ► **Implement!**
Implement this grammar in ANTLR4. Write test files. In particular, verify the fact that '*' has higher priority on '+.' Is '+' left or right associative ?

EXERCISE #6 ► **Evaluating arithmetic expressions with** ANTLR4 **and** PYTHON
Based on the grammar you just wrote, build an expression evaluator (in the `ariteval` directory). You can proceed incrementally as follows:

- Attribute the grammar to evaluate arithmetic expressions. For the moment, consider that all ids have "value 42".
- Execute your grammar against expressions such as `1+(2*3);`.
- Augment the grammar to treat lists of assignments. You will use PYTHON dictionaries to store values of ids when they are defined. The assignements can be separated by line breaks.
- Execute your grammar against lists of assignements such as `x=1;2+x;`. You should decide what to do when you encounter a variable name that is not already defined (assigned).

Here is an idea of the expected outputs:

Input	Output (on stdout)
`1;`	1 = 1
`-12;`	-12 = -12
`12;`	12 = 12
`1 + 2;`	1+2 = 3
`1 + 2 * 3 + 4;`	1+2*3+4 = 11
`(1+2)*(3+4);`	(1+2)*(3+4) = 21
`a=1+4/1;`	a now equals 5
`b + 1;`	b+1 = 43
`a + 8;`	a+8 = 13
`-1 + x;`	-1+x = 41

In the repository, we provide you a script that enables you to test your code. Just type:

```
make tests
```

and your code will be tested on all files in `testfiles/` We will use the same exact script to test your code (but with our own test cases!).

This exercise is due by email to `aurore.alcolei@ens-lyon.fr` and `valentin.lorentz@ens-lyon.fr` before Sunday, Sept 24th 8pm. Type `make tar` to obtain the archive to send (change your name in the Makefile before!)

EXERCISE #7 ► **Bonus : prefixed expressions**
Consider prefixed expressions like `* + * 3 4 5 7` and assigments of such expressions to variables:
`a=* + * 3 4 5 7`. Identifiers are allowed in expressions.

- Give a grammar that recognizes lists of such assigments. Encode it in ANTLR4. Test.
- Use grammar rules to construct infix assigments during parsing: the former assigment will be transformed into the *string* `a=(3 * 4 + 5)*7`. Be careful to avoid useless parentheses.
- Modify the attribution to verify that the use of each identifier is done after his first definition.