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.

WARNING Exercice 4 is evaluated (personal work only). Deadline October 5th.

2.1 User install for ANTLR4 and ANTLR4 Python runtime

User installation steps:

```
mkdir ~/lib
cd ~/lib
wget http://www.antlr.org/download/antlr-4.5.3-complete.jar
pip install antlr4-python2-runtime --user
```

Then in your `.bashrc`:

```
export CLASSPATH=".:/home/mylogin/lib/antlr-4.5.3-complete.jar:$CLASSPATH"
alias antlr4='java -jar /home/mylogin/lib/antlr-4.5.3-complete.jar'
alias grun='java org.antlr.v4.gui.TestRig'
```

Then, download and uncompress the archive available on the course webpage.

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/master/doc/lexer-rules.md`
- Parser rules (grammars) `https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md`

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

```
java -jar /link/to/antlr-4.5.3-complete.jar -Dlanguage=Python2  filename.g4
```

or if you modified your `.bashrc` properly:

```
\verb!antlr4 -Dlanguage=Python2  filename.g4!
```

2.3 Simple examples with ANTLR4

EXERCISE #1 ▶ **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:

```
java -jar /link/to/antlr-4.5.3-complete.jar Exemple1.g4
javac *.java
```

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

```
grun Example1 tokens
```

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

```
% ./exemple
1+1^D
% ./exemple
)+^D
line 1:0 token recognition error at: ')'
line 1:2 token recognition error at: '}'
%
```

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 :

```
make
make run
```

test the same expressions. Observe the PYTHON file.

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, make sure that you understand what is going on. You can ask the Teaching Assistant, or another student, for advice.

EXERCISE #2 ► **Well-founded parenthesis**
Write a grammar and files to accept any text with well-formed parenthesis ')' and '['.

EXERCISE #3 ► **CSV - Skip if you are late**
A csv (for comma-separated values[2]) file:
- is plain text using a character set such as ASCII, various Unicode character sets (e.g. UTF-8), EBCDIC, or Shift JIS,
- consists of records (typically one record per line),
- with the records divided into fields separated by delimiters (typically a single reserved character such as comma, semicolon, or tab; sometimes the delimiter may include optional spaces),
- ...

Write a grammar and files to recognize csv files.

For the sake of the exercise, you'll only consider one type of separator (comma ",“). You are not obliged to be able to match strings like `"Super, ""luxurious"" truck"`.

EXERCISE #4 ► **If then else ambiguity - Skip if you are late**
We give you the following grammar for imbricated "ifs" :

[1] Lexer Grammar in ANTLR4 jargon
[2] see `https://en.wikipedia.org/wiki/Comma-separated_values`

```
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". A simple solution if we want to avoir this problem consists in inventing "if then elif ...".

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):

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

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. 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 suchs 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;`	Value is 1
`12;`	Value is 12
`1 + 2;`	Value is 3
`1 + 2 * 3 + 4;`	Value is 11
`(1+2)*(3+4);`	Value is 21
`a=1+4;`	Variable 'a' has been defined with value 5
`b + 1;`	Error
`a + 8;`	Value is 13

This exercise is due by email to `lionel.morel@insa-lyon.fr` and `aurelien.cavelan@ens-lyon.fr` before Wednesday, October 5th 8pm

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.