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.

First download and uncompress the archive available on the course webpage.

WARNING Exercice 4 is evaluated (personal work only). Deadline Sunday, October 9th, 2016, 23h59, on TOMUSS.

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

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
```

2.3 Simple examples with ANTLR4

EXERCISE #1 ▶ **Demo files**
Work your way through the three 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
```

[1] Lexer Grammar in ANTLR4 jargon

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

```
grun Exemple1 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.

```
% grun Exemple1 tokens
1+1^D
% grun Exemple1 tokens
)+^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 (cf. exemple5)

ex1bis : Same with a PYTHON file driver (you must edit the `Makefile` to modify the `ANTLR4` variable according to your settings) :

```
make
make run
```

Test the same expressions, and 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 ▶ **If then else ambiguity - Skip if you are late**
We give you the following grammar for imbricated "ifs" :

```
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 you already saw in the second exercise session (the end of an expression is a semicolon):

$$Z \rightarrow E;$$
$$E \rightarrow E + T$$
$$E \rightarrow T$$
$$T \rightarrow T * F$$
$$T \rightarrow F$$
$$F \rightarrow id$$
$$F \rightarrow int$$
$$F \rightarrow (E)$$

Important remark! *"Parser rules start with a lowercase letter and lexer rules with an upper case."*[2]

EXERCISE #3 ► **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 #4 ► **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 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;`	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 on TOMUSS `https://tomuss.univ-lyon1.fr/`

2.5 Implicit tree walking using Listeners and Visitors

By default, ANTLR4 can generate code implemeting a Listener over your AST. This listener will basically use ANTLR4's built-in ParseTreeWalker to implement a traversal of the whole AST.

[2] `http://stackoverflow.com/questions/11118539/antlr-combination-of-tokens`

EXERCISE #5 ► **Error recovery with listeners**
Observe and play with the `Hello` grammar and its PYTHON Listener.

In the previous exercise, we have traversed our AST with a listener. The main limit of using a listener is that the traversal of the AST is directed by the walker object provided by ANTLR4. So if you want to apply transformations to parts of your AST only, using listener will get rather cumbersome.

To overcome this limitation, we can use the Visitor design pattern[3], which is yet another way to seperate algorithms from the data structure they apply to. Contrary to listeners, it is the visitor's programmer who decides, for each node in the AST, whether the traversal should continue with each of the node's children.

For every possible type of node in your AST, a visitor will implement a function that will apply to nodes of this type.

EXERCISE #6 ► **Implicit AST with visitor**
You already have implemented an expression evaluator with "right actions". Observe and play with the `Arit` grammar and its PYTHON Visitor.

Also note the `#blabla` pragmas in the g4 file. They are here to provide ANTLR4 a name for each alternative in grammar rules. These names are used in the visitor classes.

[3] `https://en.wikipedia.org/wiki/Visitor_pattern`