

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.

Todo in this lab:

- Install and play with ANTLR
- Code an arithmetic evaluator, which is due on tomuss on October 31, 2017.

EXERCISE #1 ► Lab preparation

In the `mif08-labs` directory¹:

```
git pull
```

will provide you all the necessary files for this lab in TP02. You also have to install ANTLR4. For tests, we will use `pytest`, you may have to install it:

```
pip3 install pytest --user
```

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 add to 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
```

EXERCISE #2 ► Install

Install and test ANTLR4 with the examples of section 2.2.

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>

¹if you don't have it already, get it from <https://github.com/lauregonnord/mif08-labs.git>

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

```
java -jar /link/to/antlr-4.7-complete.jar -Dlanguage=Python3 filename.g4
```

or if you modified your `.bashrc` properly:

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

(note: `antlr4`, not `antlr` which also exists but is not the one we want)

2.3 Simple examples with ANTLR4

EXERCISE #3 ► Demo files

Work your way through the five examples in the directory `demo_files`:

ex1 with ANTLR4 + Java : A very simple lexical analysis² for simple arithmetic expressions of the form `x+3`. To compile, run:

```
java -jar /link/to/antlr-4.7-complete.jar 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).

ex1b : same with a PYTHON file driver :

```
antlr4 -Dlanguage=Python3 Example1b.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`.

²Lexer Grammar in ANTLR4 jargon

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.

From now you will write your own grammars. Be careful the ANTLR4 syntax use unusual conventions: “Parser rules start with a lowercase letter and lexer rules with an upper case.”^a

^a<http://stackoverflow.com/questions/11118539/antlr-combination-of-tokens>

EXERCISE #4 ► 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!

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 ► Test the provided code (ariteval/ directory)

To test your installs:

1. Type

```
make ; python3 arit2.py testfiles/foo01.txt
```

This should print:

```
prog = Hello
```

on the standard output.

2. Type:

```
make tests
```

This should print:

```
test_ariteval.py::TestEVAL::test_expect[./testfiles/foo01.txt] PASSED
```

To debug our grammar, we will use an interactive test (unlike `make tests` which is fully automatic): display the parse tree graphically, and check manually that it matches your expectation. To help you, we provide a `make grun-gui` target in the Makefile, that runs `grun -gui` internally. Since `grun` uses the Java target of ANTLR, we first need to remove temporarily any Python code in our `Arit2.g4` file. Comment-out (i.e. prefix with `//`) the header part of the grammar (you will uncomment it when you start writing Python code in the grammar), and replace the main rule with `prog: ID;`. You can now run:

```
make TESTFILE=testfiles/foo01.txt grun-gui
```

If you see a Java syntax error, you probably didn't comment out the Python code properly. Otherwise, you should see a graphical window containing the parse tree of `testfiles/foo01.txt` when parsed by the grammar.

EXERCISE #6 ► Implement!

In the `ariteval/Arit2.g4` file, implement **THIS** grammar in ANTLR4. Write test files and test them wrt. the grammar with `make grun-gui`, like in the previous exercise. Keep your test files for the next exercise. In particular, verify the fact that `*` has higher priority on `+`. Is `+` left or right associative ?

Important! Before you begin to implement, it is MANDATORY to read carefully until the end of the lab subject.

EXERCISE #7 ► Evaluating arithmetic expressions with ANTLR4 and PYTHON

Based on the grammar you just wrote, build an expression evaluator. You can proceed incrementally as follows (but test each step!):

- Attribute the grammar to evaluate arithmetic expressions. For the moment, launch an error for all uses of variables:

```
if ($ID.text not in idTab):
    raise UnknownIdentifier($ID.text);
```

- 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:

```
idTab[$ID.text]=...
```

The assignments can be separated by line breaks.

- Execute your grammar against lists of assignments such as `x=1;2+x;`. When you read a variable that is not (yet) defined, you have to launch the `UnknownIdentifier` error.

Here are examples of expected outputs³:

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

The parsed expression can be printed from an `expr` for instance with:

```
rule :
    expr ... {print($expr.text);}
```

³The expected behavior of your evaluator may not be completely specified. If you make a design choice, explain it in the Readme file.

EXERCISE #8 ► Test infrastructure

We provide to you a test infrastructure. 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/`⁴.

We will use the same exact script to test your code (but with our own test cases!).

A given test has the following behavior: if the pragma `# EXPECTED` is present in the file, it compares the actual output with the list of expected values (see `testfiles/test01.txt` for instance). There is also a special case for errors, with the pragma `# EXIT CODE`, that also check the (non zero) return code if there has been an error followed by an `exit` (see `testfiles/bad01.txt`).

EXERCISE #9 ► Archive

This exercise is due on TOMUSS on October, 31, 2017, before 8pm. Type `make tar` to obtain the archive to send (change your name in the Makefile before!). Your archive must also contain tests (grammar tests as well as unit tests) and a `Readme.md` with your name, the functionality of the code, how to use it, your design choices, and known bugs.

⁴To test for real test cases, you should open the `test_ariteval.py` and change some paths