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.
- Implement your own grammars. **This will be evaluated next lab!**
- Understand and extend an arithmetic evaluator (with semantic actions).
- Understand our future test infrastructure.

EXERCISE #1 ▸ **Lab preparation**
In the lab's git repository (`mif08-labs20`, if you don't have it already, see the beginning of lab1).

```
git commit -a -m "my changes to LAB1" #push is not allowed
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:

```
python3 -m pip install pytest --user
```

2.1 User install for ANTLR4 and ANTLR4 Python runtime

2.1.1 User installation

EXERCISE #2 ▸ **Install**
To be able to use ANTLR4 for the next labs, download it and install the python runtime:

```
mkdir ~/lib
cd ~/lib
wget http://www.antlr.org/download/antlr-4.8-complete.jar
python3 -m pip install antlr4-python3-runtime --user
```

Then add to your `~/.bashrc`:

```
export CLASSPATH=".:$HOME/lib/antlr-4.8-complete.jar:$CLASSPATH"
export ANTLR4="java -jar $HOME/lib/antlr-4.8-complete.jar"
alias antlr4="java -jar $HOME/lib/antlr-4.8-complete.jar"
```

Then source your `.bashrc`:

```
source ~/.bashrc
```

Tests will be done in Section 2.2.2.

2.2 Simple examples with ANTLR4

2.2.1 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 if you modified your `.bashrc` properly:

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

If you did not define the alias or if you installed the `.jar` file to another location, you may also use:

```
java -jar /path/to/antlr-4.8-complete.jar -Dlanguage=Python3  filename.g4
```

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

2.2.2 Up to you!

EXERCISE #3 ► **Demo files**
Work your way through the two examples (open them in your favorite editor!) in the directory `demo_files`:

ex1: lexer grammar and PYTHON **driver** A very simple lexical analysis[1] for simple arithmetic expressions of the form x+3. To compile, run:

```
antlr4 -Dlanguage=Python3 Example1.g4
```

This generates a lexer in `Example1.py` (you may look at its content, and be happy you didn't have to write it yourself) plus some auxiliary files. We provide you a simple `main.py` file that calls this lexer:

```
python3 main.py
```

(or type `make run`, which re-generates the lexer as needed and runs `main.py`).
To signal the program you have finished entering the input, use **Control-D** (you may need to press it twice).
Examples of runs: [`^D` means that I pressed Control-D]. What I typed is in boldface.

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

Questions:
- Reproduce the above behavior.
- Read and understand the code.
- Allow for parentheses to appear in the expressions.

[1] Lexer Grammar in ANTLR4 jargon

- What is an example of a recognized expression that looks odd (i.e. that is not a real arithmetic expression)? To fix this problem we need a syntactic analyzer (see later).
- Observe the correspondance between token names and token numbers in `<..>` (see the generated `Example1.token` file)
- Observe the PYTHON `main.py` file.

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

ex2: full grammar (lexer + parser) and PYTHON **driver** Now we write a grammar for valid expressions. Observe how we recover information from the lexing phase (for ID, the associated text is `$ID.text$`). The grammar includes Python code and therefore works only with the PYTHON driver.

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 make an analyser that:

- skips all characters but ’(’, ’)’, ’[’, ’]’ (use the parser rule `CHARS: ~[()[\]] -> skip ;` for it)
- accepts well-formed parenthesis.

Thus your analyser will accept “(hop)” or “[()](tagada)” but reject “plop]” or “[)”. Test it on well-chosen examples. *Begin with a proper copy of ex2, change the name of the files, name of the grammar, do not forget the main and the Makefile, and THEN, change the grammar to answer the exercise.*

EXERCISE #5 ► **Another grammar**
Write a grammar that accepts the language $\{a^n b^{2n}\}$. Letters other than a and b, and spaces are ignored, other symbols are rejected by the lexer.

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. You can often avoid the problem by defining a function in the Python header and then call it in the right-hand side of the rules.

2.3 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.

This exercice is a demo - no grade will be given We consider a simple grammar of non empty lists of arithmetic expressions:

$$
\begin{aligned}
S &\to Z+\\
Z &\to E;\\
E &\to E+E\\
E &\to E*E\\
E &\to F\\
F &\to int\\
F &\to (E)
\end{aligned}
$$

The object of the demo is to understand how semantic action work, and also to play with the test infrastructure we will use in the next labs.

EXERCISE #6 ► **Test the provided code (`ariteval/` directory)**
To test the provided code, just type:

1. Type

   ```
   make && python3 arit1.py testfiles/test01.txt
   ```

   This should print:

   ```
   1+2 = 3
   ```

   on the standard output.

2. Type:

   ```
   make tests
   ```

   This should print:

   ```
   test_ariteval.py::TestEVAL::test_expect[./testfiles/test01.txt] PASSED   [ 50%]
   test_ariteval.py::TestEVAL::test_expect[./testfiles/test02.txt] PASSED   [100%]
   ```

EXERCISE #7 ► **Understand the test infrastructure**
We saw in the previous exercice an example for test run. In the repository, we provide you a script that enables you to test your code. It tests files of the form `testfiles/test*.txt`. Just type:

```
make tests
```

and your code will be tested on these files.

We will use the same exact script to test your code in the next labs (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 `// EXITCODE n`, that also checks the (non zero) return code n if there has been an error followed by an `exit`.

The following exercises are due on TOMUSS. Instructions and deadline are on the course's webpage.

EXERCISE #8 ► **Write tests**
Write tests: grammar tests as well as test cases: in this lab we do not expect a large set of negative tests.

EXERCISE #9 ► **Add features**
Implement binary and unary minus. Test.

EXERCISE #10 ► **Archive**
Type `make tar` to obtain the archive to send. Your archive must also contain tests (positive tests that check the behavior of your program for valid input, and negative tests that checks that invalid input is rejected. In this lab we do not expect a large set of negative tests, and we anyway don't have a clean management of syntax errors) and a `README.md` with your name, the functionality of the code, how to use it, your design choices, and known bugs (probably less than 5 lines of text to add unless you did anything special).