Lab 1
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.
- Understand and extend an arithmetic evaluator (with semantic actions).
- Understand our future test infrastructure for our compiler.
- Deliver a code for two of these exercises **according to strict specifications.**

1.1 Getting started!

EXERCISE #1 ▶ **Git clone**
On your personal machines or on the school machines, you have to clone the git repository where all codes will be given:

```
git clone https://gricad-gitlab.univ-grenoble-alpes.fr/gonnord/cs444-labs22
```

EXERCISE #2 ▶ **Tool requirements (personal machines)**
Skip if you use the school machines, everything should be already installed! On your machines, you need to install **in this order**

- Python3, its package manager pip and pytest and other stuff

  ```
  sudo apt install python3 python3-pip
  python3 -m pip install pytest pytest-cov pytest-xdist pyright --user
  ```

  (pyright is not mandatory)

- An ANTLR4 python runtime (≥ 4.8), let's use 4.9.2:

  ```
  python3 -m pip install antlr4-python3-runtime==4.9.2 --user
  ```

- The adequate ANTLR4 jar file:

  ```
  mkdir ~/lib     # or anywhere else.
  cd ~/lib
  wget https://www.antlr.org/download/antlr-4.9.2-complete.jar
  ```

- You may need to install java on your machine:

  ```
  sudo apt install default-jre
  sudo apt install default-jdk
  ```

EXERCISE #3 ▶ **System paths (all machines), only todo ONCE**

Adapt your `~/.bashrc` (or `Documents_distants/.bashrc`) to your configuration [1], for instance on my machine I have:

[1] on school machines use `/opt/antlr-4.9.2-complete.jar`

```
export CLASSPATH=".:$HOME/lib/antlr-4.9.2-complete.jar:$CLASSPATH"
export ANTLR4="java -jar $HOME/lib/antlr-4.9.2-complete.jar"
alias antlr4="java -jar $HOME/lib/antlr-4.9.2-complete.jar"
```

Then source your `.bashrc`:

```
source ~/.bashrc
```

If ANTLR4 is set correctly, then typing:

```
$ antlr4
```

in your terminal should output a documentation.

1.2 Simple examples with ANTLR4

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

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

1.2.2 Up to you!

EXERCISE #4 ▶ **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[2] 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 (this one is hand-written and readable):

```
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
```

[2] Lexer Grammar in ANTLR4 jargon

```
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.
- 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] `https://stackoverflow.com/questions/11118539/antlr-combination-of-tokens`

EXERCISE #5 ▶ **Well-founded parenthesis**
Write a grammar and files to make an analyser that:

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

Thus your analyser will accept "(hop)" or "[()](tagada)" but rejects "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 #6 ▶ **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.

1.3 Grammar Attributes (actions), `ariteval/` directory

Until now, our analyzers are passive oracles, i.e. 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 is processed/recognized. This piece of code is called *semantic action* and computes attributes of non-terminals.

We consider a simple (commented) grammar of non empty lists of arithmetic expressions:

$P \rightarrow S+$ // + denotes a list and EOF is implicit
$S \rightarrow E;$ // expressions are followed by a semicolon
$E \rightarrow E+E$
$E \rightarrow E*E$
$E \rightarrow F$
$F \rightarrow int$ // int denotes any constant int value provided by the lexer
$F \rightarrow (E)$

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

EXERCISE #7 ▶ **Test the provided code (`ariteval/` directory)**
First, have a look on the `README.md` file which gives you some information. You will have to edit it later. Now, have a look onto the grammar in file `Arit.g4`. Remark how each grammar rule has been implemented. Each grammar rule is followed by a **semantic action** at its right-hand side.

To test the provided code, just type:

1. Type

   ```
   make && python3 arit.py tests/test-plus.txt
   ```

   This should print:

   ```
   1+2 = 3
   ```

   on the standard output. **It seems that you have been given a calculator code !**

2. Type:

   ```
   make test
   ```

   This should print:

```
test_ariteval.py::TestEVAL::test_expect[./tests/test-mult.txt] FAILED                                [ 50%]
test_ariteval.py::TestEVAL::test_expect[./tests/test-plus.txt] PASSED                                [100%]

================================================ FAILURES =================================================
_________________________________ TestEVAL.test_expect[./tests/test-mult.txt] _________________________________

self = <test_ariteval.TestEVAL object at 0x7f5be113a100>, filename = './tests/test-mult.txt'

    @pytest.mark.parametrize('filename', ALL_FILES)
    def test_expect(self, filename):
        expect = self.get_expect(filename)
        eval = self.evaluate(filename)
>       assert expect == eval
E       AssertionError: assert testresult(ex...'1+3*2 = 7\n') == testresult(exi...1+3*2 = 43\n')
E         At index 1 diff: '1+3*2 = 7\n' != '1+3*2 = 43\n'
E         Full diff:
E         - testresult(exitcode=0, output='1+3*2 = 7\n')
E         ?                                    ^
E         + testresult(exitcode=0, output='1+3*2 = 43\n')
E         ?                                    ^^

test_ariteval.py:24: AssertionError
================================== 1 failed, 1 passed in 0.18 seconds ==================================
```

   This test means that you have been given only a partial solution ! Indeed, the semantic action for * given in the skeleton (Arit.g4) is obviously buggy. As the diagnosis suggests, the skeleton returns 43 when it should return 7.

EXERCISE #8 ▶ **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 [3]. It tests files of the form `tests/test*.txt`. Just type:

```
make test
```

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 `tests/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 (two) exercises are due on Chamilo. Instructions and deadline on Chamilo too.

EXERCISE #9 ▶ **Play with the testsuite**
As we saw above, the skeleton is buggy, and we only have two test cases.

1. Fix the code of the semantic action for `*` in `Arit.g4`, re-run `make test` and check that you have no failure.
2. Write one test, in `./tests/test-mult-parents.txt` that checks the proper evaluation of the expression `(1+3)*2`, to make sure you understand the test infrastructure.

In an ideal world, we should also check the behavior in case of syntax error, but ANTLR's error-recovery mechanism makes this a bit too tricky, so we won't do it in this lab.

EXERCISE #10 ▶ **Add features**
Implement binary and unary minus. Test. Be careful with operators precedence and associativity **to simplify, you can make as if - is prioritary to +**. Unary minus can apply to any expression (`--1` is accepted, with the same meaning as Python). **Of courses, you will have to write adequate tests for these new features.**

1.4 Bonus !

If you have finished in advance, we provide you a bonus exercice in the `bonus/` directory.

[3] This infrastructure is based on pytest, you can have a quick look at the `test_*.py` to understand it. These files do the following: for each test file, execute the code on it, and compare its output with the specification written at the end in comments.