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.
- Write a first evaluator under the form of semantic actions in ANTLR4.

EXERCISE #1 ▶ **Lab preparation**
In the `cap-labs21` directory[1]:

```
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
```

If the command above fails, you need to install `pip` on your machine and re-launch the command, on Ubuntu this is done with `sudo apt install python3-pip`, see `https://pip.pypa.io/en/stable/installation/` otherwise.

2.1 Typing in python with Pyright

Python is originally a dynamically typed language, i.e. types are associated with values at runtime, but no check is done statically. Recent versions of Python, however, allow adding static typing annotations in the code. You don't *need* to add these annotations, but having them allows static typecheckers like Mypy or Pyright to find errors in the code before running it. In this course, we will use Pyright. You don't need to understand all the details of typing annotations, but you need to get familiar with them to understand the code provided to you.

If needed, install pyright. It is provided in the archive and the Docker image, so you probably have nothing to do.

EXERCISE #2 ▶ **Basic type checking**
Consider the following code (available as `TP01/python/typecheck.py` in your repository):

```python
# Typing annotations for variables:
# name: type
int_variable: int
float_variable: float
int_variable = 4.2 # Static typing error, but no runtime error
float_variable = 42.0 # OK
float_variable = int_variable # OK


# Typing annotations for functions (-> means "returns")
def int_to_string(i: int) -> str:
    return str(i)


print(int_to_string('Hello')) # Static typing error, but no runtime error
print(int_to_string(42) / 5) # Both static and runtime error
```

[1] if you don't have it already, get it from `https://github.com/Drup/cap-labs21.git`

Run the code in the Python interpreter:

```
python3 typecheck.py
```

Check for static typing errors using `pyright`:

```
pyright typecheck.py
```

Note how static typechecking finds more error in your code, and possibly saves you a lot of debbugging time. Try modifying the code to get rid of typing errors.

EXERCISE #3 ▶ **Type unions**
Play with (execute, typecheck, read the comments) the code `TP01/python/type_unions.py` in your repository. `Union[int, float]` means "either an `int` or a `float`". `List[NUMBER]` means "a list whose elements are numbers".

2.2 User install for ANTLR4 and ANTLR4 Python runtime

2.2.1 User installation

EXERCISE #4 ▶ **Install**
To be able to use ANTLR4 for the next labs, download it and install the python runtime:

```
mkdir ~/lib
cd ~/lib
wget https://www.antlr.org/download/antlr-4.9.2-complete.jar
python3 -m pip install antlr4-python3-runtime==4.9.2 --user
```

Then add to your `~/.bashrc`:

```
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
```

Tests will be done in Section 2.3.2.

2.3 Simple examples with ANTLR4

2.3.1 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-4.9.2-complete.jar -Dlanguage=Python3  filename.g4
```

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

2.3.2 Up to you!

EXERCISE #5 ▸ **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:

```
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.
- 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 #6 ▸ **Well-founded parenthesis**
Write a grammar and files to make an analyser that:

[2] Lexer Grammar in ANTLR4 jargon

- 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.*

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.

EXERCISE #7 ► **Towards analysis: If then else ambiguity**[3] **- Skip if you are late**
We give you the following grammar for nested "ifs" (`ITE/` directory):

```
grammar ITE;

prog: stmt EOF;

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". There are other ways of getting around this problem explicitly:

- by changing the syntax: add explicit parentheses, or other block marks ("if .. then .. end else .. end")
- by disambiguating the grammar:
  ```
  ifStmt : 'if' ID ifStmt | 'if' ID ifThenStmt 'else' ifStmt
  ifThenStmt : 'if' ID ifThenStmt 'else' ifThenStmt
  ```
- by using the ANTLR4 "lookahead" concept.
  ```
  ifStmt : 'if' ID stmt ('else' stmt | {self.input.LA(1) != ELSE}?);
  ELSE : 'else';
  ```

more on `https://en.wikipedia.org/wiki/Dangling_else`

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

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.

[3] Also known as "dangling else"

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 #8 ▶ **Test the provided code (`ariteval/` directory)**

To test your installs:

1. Type

   ```
   make ; python3 arit2.py tests/hello01.txt
   ```

   This should print:

   ```
   prog = Hello
   ```

   on the standard output.

2. Type:

   ```
   make tests
   ```

   This should print:

   ```
   test_ariteval.py::TestEVAL::test_expect[./tests/hello01.txt] PASSED
   ```

To debug our grammar, we will use an interactive test (unlike `make tests` which is fully automatic): display the parse tree in Lisp format, and check manually that it matches your expectation.

To help you, we provide a `make print-tree` target in the Makefile, which you can run:

```
make TESTFILE=tests/hello01.txt print-tree
```

prints:

```
(prog Hello)
```

EXERCISE #9 ▶ **Second test: only if you have time and a full Java distribution**

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 temporary 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=tests/hello01.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 `tests/hello01.txt` when parsed by the grammar :

EXERCISE #10 ▸ **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` or `make print-tree`, like in the previous exercises. 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 #11 ▸ **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 with binary operators in $\{+,-,*,/\}$ and unary operator $-$(minus) with the usual priorities. / will be the integer division such that $4/3 = 1$. For the moment, launch an error for all uses of variables:

  ```
  if $ID.text not in idTab: # Always true, for now
      raise UnknownIdentifier($ID.text);
  ```

- Execute your grammar against expressions such as `1+(2*3);`.
- Augment the grammar to treat lists of assignments (and expressions). You will use PYTHON dictionaries to store values of ids when they are defined:

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

  Line breaks should now be allowed between assignments (and expressions).
- Execute your grammar against lists of assignements such as `x=1;2+x;`. When you read a variable that is not (yet) defined, you have to launch the `UnknownIdentifier` error.
- Division by zero: do what you want but explain (see below how to handle automated test for errors).

Here are examples of expected outputs [4]:

Input	Output (on stdout)
`1;`	1 = 1
`-12;`	-12 = -12
`12;`	12 = 12
`1 + 2;`	1+2 = 3
`3 - 2;`	3-2 = 1
`1 + 2 * 3 + 4;`	1+2*3+4 = 11
`(1+2)*(3+4);`	(1+2)*(3+4) = 21
`a=1+4/1;`	a now equals 5
`b + 1;`	b+1 = 43
`a + 8;`	a+8 = 13
`-1 + x;`	-1+x = 41
`-(5+a);`	-(5+a) = -10
`3 - (-4);`	3-(-4) = 7
`3 - -4;`	3–4 = 7
`4/3;`	4/3 = 1

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

[4] The expected behavior of your evaluator may not be completely specified. If you make a design choice, explain it in the `README.md` file. (div by 0, for instance)

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

EXERCISE #12 ▶ **Test infrastructure**

We provide to you a test infrastructure. In the repository, we provide you a script that enables you to test your code. For the moment it only tests files of the form `tests/foo*.txt`.

Just type:

```
make tests
```

and your code will be tested on these files.

To test on more relevant tests, you should open the `test_ariteval.py` and change some paths.

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 `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` (see `tests/bad01.txt`).

EXERCISE #13 ▶ **Archive and test deposit**

We do not ask you for a mandatory deposit. However we strongly encourage you to have an operational evaluator and above all, understand the test infrastructure.

We nevertheless encourage you to do a test deposit:

- Type `make tar` to obtain the archive to send. Your archive must also contain test cases in the `tests/` directory 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).
- Go to `https://etudes.ens-lyon.fr/course/view.php?id=4814` and deposit your archive.