Lab 2
Lexing and Parsing with ANTLR4

Objective

- Discover typing in python with Pyright.
- 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.
- Optionally, deposit your work on `https://etudes.ens-lyon.fr` and get feedback (see instructions at the end).

EXERCISE #1 ▶ **Lab preparation**
In the `cap-lab24` directory[1]:

```
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`, and for printing trees we will use graphviz, you may have to install them:

```
sudo apt install graphviz
python3 -m pip install pytest pytest-cov pytest-xdist graphviz
```

If the pip command 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 code found in `TP02/python/typecheck.py` in your repository.

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.

[1] if you don't have it already, get it from `https://github.com/Drup/cap-lab24.git`

EXERCISE #3 ▶ **Type unions**

Play with (execute, typecheck, read the comments) the code found in `TP02/python/type_unions.py` in your repository.

The syntax `int | float` (or equivalently `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**

If you are using the docker image or the precompiled RiscV toolchain, you do not need to install ANTLR4 as it is already included.

Otherwise, 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-4.13.1-complete.jar
$ python3 -m pip install antlr4-python3-runtime==4.13.1
```

Then add the follwing lines to your `~/.bashrc`:

```
export ANTLR4="java -jar $HOME/lib/antlr-4.13.1-complete.jar"
alias antlr4="java -jar $HOME/lib/antlr-4.13.1-complete.jar"
```

If you are using the precompiled toolchain, add the following instead:

```
export ANTLR4="java -jar /opt/antlr-4.13.1-complete.jar"
alias antlr4="java -jar /opt/antlr-4.13.1-complete.jar"
```

Then restart you terminal, or source your `.bashrc`:

```
$ source ~/.bashrc
```

You may need to install java on your machine:

```
$ sudo apt install default-jre
```

Tests will be done in Section 2.4.

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

```
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-4.13.1-complete.jar -Dlanguage=Python3  filename.g4
```

2.3.2 Up to you!

EXERCISE #5 ▶ **Demo files**
Work your way through the three 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
```

(or type `make`).

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
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 in `Example1.g4`. Observe the outputs above.
- Modify the source file to allow for parentheses to appear in the input.
- 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 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 in `Example2.g4` how we recover information from the lexing phase in the ID rule (for ID, the associated text is `$ID.text`). More generally, we can associate to each production a piece of Python code that will be executed each time the production is reduced. This piece of code is called a *semantic action* and computes attributes of non-terminals.

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.

[2] Lexer Grammar in ANTLR4 jargon

ex3: full grammar with attributes and PYTHON driver We take the exact same grammar as for ex2, but modify its semantic actions so as to compute the number of operators in an expression. Observe how to define an attribute in the grammar with `expr returns [type name]`, and how to propagate it through the syntax with actions.

As before, once you understand these files, go on with the other exercises.

> 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 parentheses**
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 parentheses.

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 (use* `make clean`*), 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 #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 'then' thenstmt=stmt ('else' elsestmt=stmt)?;


ID : [a-zA-Z]+;
WS : [ \t\r\n]+ -> skip ; // skip spaces, tabs, newlines
```

Find a way (with the right semantic actions) to test if:

```
if x then if y then a else b
```

is parsed as:

```
if x then (if y then a else b)
```

or

```
if x then (if y then a) else b
```

Is it the same with this grammar as with real-life C-like languages?

Notice the `thenstmt=stmt` syntax to give unique names to the two `stmts` of the above `ifStmt` grammar rule.

There are more explicit ways of getting around the dangling else problem by adapting the grammar. For more details see `https://en.wikipedia.org/wiki/Dangling_else`

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

Until now, our analyzers were 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.

[3] Also known as "dangling else"

Important remark As Python is sensitive to indentation, there might be some issues when writing semantic actions on several lines. You can often avoid the problem by defining a function in the Python header and then calling it in the right-hand side of the rules.

The goal of this section is to implement a simple arithmetic evaluator, starting from the basic infrastructure given in the `ariteval/` directory.

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

To test your installs:

1. Type

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

   This should print:

   ```
   prog = Hello
   ```

   on the standard output.

2. Type:

   ```
   make test
   ```

   This should print:

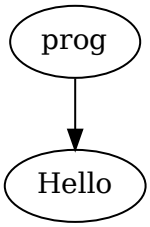
   ```
   test_ariteval.py::TestEVAL::test_expect[./tests/hello01.txt] PASSED
   ```

To debug our grammar, we will use an interactive test (unlike `make test` 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
```

It opens a window with a graphical representation of the parse tree:



The result is also stored in `tree.dot.pdf`.

Alternatively, replace `print-tree` with `print-lisp` in the above command to display a textual (LISP) representation of the parse tree:

```
(prog Hello)
```

We will now implement the following grammar (an expression followed by a semicolon):

$$
\begin{aligned}
Z &\rightarrow E; \\
E &\rightarrow E + T \\
E &\rightarrow T \\
T &\rightarrow T * F \\
T &\rightarrow F \\
F &\rightarrow id \\
F &\rightarrow int \\
F &\rightarrow (E)
\end{aligned}
$$

Figure 2.1: A grammar for arithmetic expressions

EXERCISE #9 ▶ **Implement!**
In the `ariteval/Arit.g4` file, implement the grammar from figure 2.1 in ANTLR4. Write test files and test them wrt. the grammar with `make print-tree`. 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 #10 ▶ **Evaluating arithmetic expressions with** ANTLR4 **and** PYTHON
Attribute the grammar you just wrote to evaluate arithmetic expressions with binary operators in $\{+,-,*,/\}$ with the usual priorities and print the result; / will be the integer division such that $4/3 = 1$. For the moment, throw an error for all uses of variables:

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

To test your grammar, run it against expressions such as `1+(2*3);`. In this case the expected output is `1+(2*3) = 7`.

In case of division by zero, do what you want but explain (see the next exercise for how to handle automated test for errors).

EXERCISE #11 ▶ **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/hello*.txt`.

To test your code on these files, just type:

```
make test
```

To test on more relevant files, you should open `test_ariteval.py` and change the value of `ALL_FILES`.

We will use the exact same 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 #12 ▶ **Adding features!**

- 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).

- Write tests for lists of assignements such as `x=1;2+x;`. When you read a variable that is not (yet) defined, you have to raise the `UnknownIdentifier` exception.
- Optionally, implement binary and unary minus. Be careful with operators precedence and associativity. Unary minus can apply to any expression (`--1` is accepted, with the same meaning as in Python).

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 expression, for instance with:

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

[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 (division by 0, for instance).