



Week 2 Exercises (Lexical Analysis with Antlr4 and Python3)

Before starting – *if you are working from your own machine*, make sure you have installed all the required software for these exercises: (<https://docs.anaconda.com/anaconda/install/>).

Use Linux!

The exam will be in Linux, so you should use it in every lab.

To use Antlr4 follow these instructions:

Open a terminal and enter the following command:

```
pip3 install antlr4-python3-runtime
```

This will install the antlr4 bindings that you need for Python.

If you experience difficulties, like pip3 not being recognised as a command (e.g. if you are on your own virtual machine), it means it is not installed. To install it, enter the following commands one at a time:

```
sudo apt-get update
```

```
sudo apt-get install python3-pip
```

The try installing Antlr4 again:

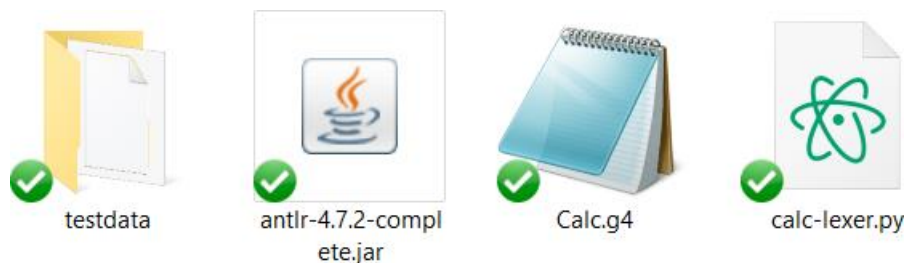
```
pip3 install antlr4-python3-runtime
```

This lab is designed to familiarise you with your working environment and the tools we are using this term for your coursework. You are encouraged to work freely on your coursework during labs, using these exercises as familiarisation and practice.

(Task 1) Regular Expressions in Antlr4 (Warmup Task)

In this short warmup task, you will work with some starter files and develop a lexical analyser using Antlr4 and Python3. We are going to work on a made-up language called Calc – which is designed to do some basic arithmetic with variables and numbers. Today we will work on the lexical analysis part, and in future weeks we will develop the parsing, semantic checking and code generation parts.

1.1 Download the **Calc Lexer Starter Files** under week 2 in this unit's Moodle area and unzip them to a suitable location. When you have unzipped them, you should see the following files:



The **testdata** folder contains some files to help you test your lexer. The **antlr-4.7.2-complete.jar** is the language recognition tool Antlr4, which you will use to create a lexer. The **Calc.g4** is a text file in which you will define the grammar for our language. The **calc-lexer.py** is a Python script in which you will load the lexer created by Antlr and process some text files in the **testdata** folder.





1.2 Open Atom (or some other suitable code/text editor) and load the **Calc.g4** file.

```
1  grammar Calc;
2
3  ADD : '+';
4  LBRACE : '(';
5  RBRACE : ')';
6
7  fragment ALPHA : [a-z];
8  VAR : ALPHA+;
9
10 WS : ' ' + -> skip;
11
12 calculator : LBRACE RBRACE;
```

On line 1, we use an Antlr command to define a grammar called Calc. The first thing to note is that the **name of the grammar must match the name of the .g4 file** (i.e. file must be called Calc.g4). If the filename does not match the grammar name, Antlr will not process it.

From line 3 until line 10 there are some rules which define the words (tokens) of our language. They are all upper case, because Antlr recognises upper case rules as lexer rules.








On line 12 there is a single rule which defines a valid Calc program. The rule is all lower case, because Antlr recognises lower case rules as parser rules.

1.3 Open a terminal and navigate to the folder containing your Calc.g4 and Antlr4 jar files. Then enter the following command:

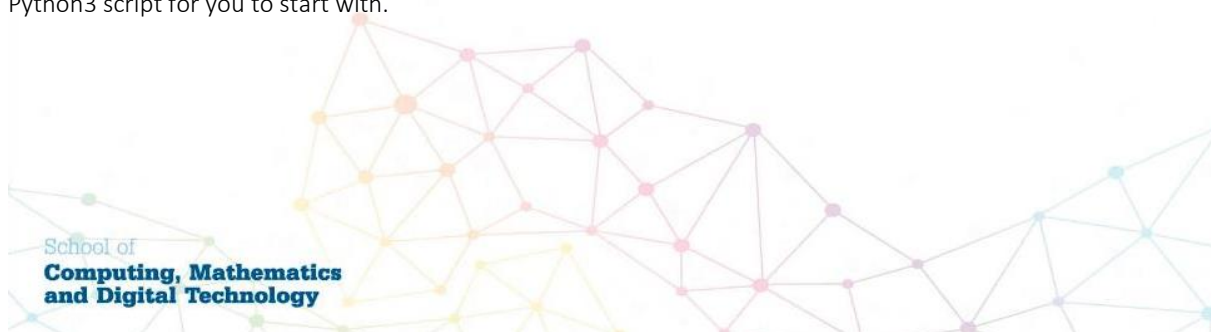
```
java -Xmx500M -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool -Dlanguage=Python3 Calc.g4
```

There should be no errors and a bunch of files should have been created for you – if there are, and you cannot resolve them, ask the tutor for help.

The following files should have been created for you:

-  Calc.interp
-  Calc.tokens
-  CalcLexer.interp
-  CalcLexer.py
-  CalcLexer.tokens
-  CalcListener.py
-  CalcParser.py

For now, we are only interested in the **CalcLexer.py** file. This is a Python3 class file which can be imported to a regular Python3 script. It is a lexer that has been created using your .g4 grammar file. I have already created a Python3 script for you to start with.





1.4 Open the calc-lexer.py file in **Spyder** or **Atom** (or some other code/text editor).

```
1 import antlr4 as ant
2 from CalcLexer import CalcLexer
3
4 filein = open('testdata/test_01.calc', 'r')
5 lexer = CalcLexer(ant.InputStream(filein.read()))
6
7 token = lexer.nextToken()
8 if (token.type != -1):
9     print(lexer.symbolicNames[token.type])
```

On lines 1 and 2 we import the required Antlr library and the lexer we just created using the Antlr tool.

On line 4 we open one of the test files to use as our input source.

On line 5 we create a token stream using our test file as input.

On line 7 we extract the first token from the token stream.

On line 9 we print the name of the token (as defined in your .g4 grammar file), which can be extracted from the token.type field and the lexer.symbolicNames list.

If there are no more tokens in the stream, then the token.type will be equal to -1, hence the if statement on line 8.

Execute this Python code and observe the output:

LBRACE

Here you see that the first token was the LBRACE token. Check the source file, “testdata/test_01.calc” to confirm:

```
1 (x + y)
```

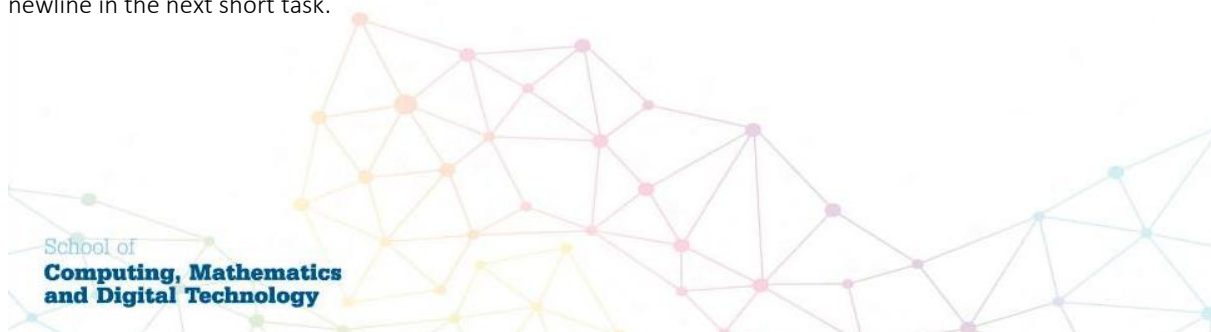
Correct! The first token is a left bracket.

1.5 Now you are familiar with this process, your first task is to write a loop which iterates over all tokens and prints the name of the token. For the test_01. calc file you should see the following output:

```
LBRACE
VAR
ADD
VAR
RBRACE
line 1:7 token recognition error at: '\n'
```

Do not worry about the token recognition error – this is just because the grammar is incomplete.

You will notice that our entire input string has been converted into tokens, and they appear to be correctly matched – a left brace, followed by a variable, then an add op, then a variable, then a right brace. We will deal with the newline in the next short task.





1.6 Now are familiar with the process of generating a lexer using the Antlr tools, and converting raw text into a token stream, let's fix the error we got in task 1.5. This requires modifying the rules in the .g4 grammar file.

The problem is that there is a newline character `\n` at the end of our first line of Calc code. One of the jobs of a lexer is to throw away (ignore) whitespace which includes formatting characters like spaces, new lines, tabs, etc. On line 10 of the .g4 grammar file there is a rule,

```
10 WS : ' '+ -> skip;
```

The WS is the name of the rule, then everything to the right of the `:` and to the left of the `->` is a regular expression. It is a rather simple regular expression which recognises spaces. The plus operator means one or more, meaning that there can be any number of consecutive spaces but there must be at least 1. To the right of the `->` there is an instruction **skip** which tells Antlr not to create a token in the lexer and to just discard the input.

We want to extend this rule so that new line characters are also skipped. We can do that using an or operator, which is the `|` symbol. Change the rule to the following:

```
10 WS : (' ' | '\n')+ -> skip;
```

This regular expression says that WS can be any number (at least 1) of consecutive spaces or new lines. Before we apply the plus operator, we use parentheses to group the two expressions, else the plus would bind to the new line.

Here is an alternative way to write the same rule:

```
10 WS : [ \n]+ -> skip;
```

Above, square brackets denote a list of alternatives. Within the square brackets there is a space and a newline, which means exactly the same as the previous rule we wrote. It can be convenient to use square bracket notation, but either is fine.

Now, build your lexer using the Antlr tool:

```
java -Xmx500M -cp antlr-4.7.2-complete.jar org.antlr.v4.Tool -Dlanguage=Python3 Calc.g4
```

Then try running your Python script again. You should get the following output, with no errors:

```
LBRACE
VAR
ADD
VAR
RBRACE
```





(Task 2) Interpreting Formal Grammar (Warmup Task)

In this task you will experience writing grammar in Antlr from a formal grammar, like the one in your coursework document. The grammar for the Calc language is given below along with a notation table – use the notation table to interpret the symbols. The grammar is written in a similar syntax to regular expressions – the plus, asterisk and pipe operators do the same thing in the grammar as in regular expressions (see notation table).

Grammar Notation Table

Notation	Description
<foo>	Non-terminal symbol
foo	Terminal symbol
x*	Zero or more x
x+	One or more x
	OR

Calc Grammar

<calculator>	→	(<statement>* <expr>)
<expr>	→	<var>
		<number>
		<expr> <op> <expr>
		(<expr>)
<op>	→	+ - * /
<statement>	→	<var> = <expr> ;
<alpha>	→	a b c ... z A B C ... Z _
<var>	→	<alpha>+
<digit>	→	0 1 2 3 ... 9
<number>	→	<digit>+

Refer to the grammar above. Non-terminal symbols are like variables/rules that can be replaced with other rules, while terminal symbols are those that can only be replaced by the raw text of the language. The grammar above represents the syntax of the language, including both parser rules and lexer rules. Which rules are parser rules, and which are lexer rules?

To build a good lexical analyser, we need to recognise common patterns of text and replace them with tokens, so that the parser can more efficiently parse a valid Calc program. Generally speaking, the lexer rules are the valid ‘words’ of the language – variable names, numbers, brackets, operators. It is up to you to determine which ones to implement in the lexer and which to implement in the parser. Here are some general rules paraphrased from page 79 of *The Definitive ANTLR 4 Reference* (Terence Parr, 2012):

1. Match and discard anything in the lexer that the parser does not need to see at all. Recognize and toss out things like whitespace and comments for programming languages.
2. Match common tokens such as identifiers, keywords, strings, and numbers in the lexer.
3. Lump together into a single token type those lexical structures that the parser does not need to distinguish. For example, if our application treats integer and floating-point numbers the same, then lump them together as token type **number**.
4. Lump together anything that the parser can treat as a single entity.
5. On the other hand, if the parser needs to pull apart a lump of text to process it, the lexer should pass the individual components as tokens to the parser.





2.1 Using the grammar above, write the lexer rule (regular expression) for a **number** in your .g4 grammar file, then rebuild your lexer using the Antlr tools as in previous tasks, and execute the decaf-lexer.py Python file on the **test_02.calc** test file. If successful, you should see the following output:

```
LBRACE
NUMBER
ADD
NUMBER
RBRACE
```

If you are unsure about regular expressions, refer to the table below or an online reference like:

https://en.wikipedia.org/wiki/Regular_expression#Syntax

Regular Expression Notation Table

Notation	Description
'a'	Matches the single character 'a'
'a' 'b' 'c'	Matches the concatenation of 'a' 'b' and 'c' e.g. 'abc'
[a]	Matches the single character 'a'
[abc]	Matches any single character 'a' or 'b' or 'c'
[a-z]	Matches any single character between 'a' and 'z'
'a'?	Matches zero or one 'a' e.g. "" or 'a'
'a'+	Matches one or more of consecutive 'a' e.g. 'aaaaaaa'
'a'*	Matches zero or more of consecutive 'a' e.g. "" or 'aaaaaaa'
[ab] [01]	Matches any single character 'a' or 'b' followed by any single character '0' or '1'
[ab] [01]*	Matches any single character 'a' or 'b' followed by zero or more of consecutive '0' or '1'

2.2 Using the grammar above, modify and complete the lexer rule (regular expression) for an **var** in your .g4 grammar file, then rebuild your lexer using the Antlr tools as in previous tasks, and execute the decaf-lexer.py Python file on the **test_03.calc** test file. If successful, you should see the following output:

```
VAR
VAR
VAR
VAR
VAR
VAR
VAR
VAR
VAR
```

2.3 Using the grammar above, modify and complete the lexer rules (regular expressions) for an all other identifiable tokens (given the definitions above) in your .g4 grammar file. Then rebuild your lexer using the Antlr tools as in previous tasks, and execute the decaf-lexer.py Python file on the remaining test files.

Do not implement the parser rules – we will do this in future labs.



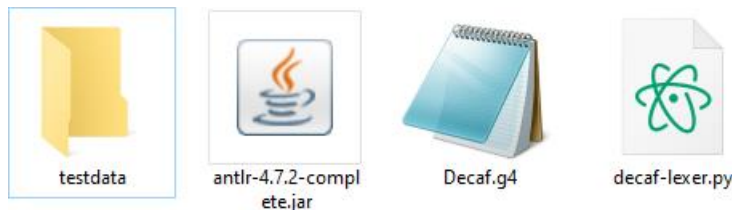


(Task 3) Decaf Lexer (Coursework)

Now that you have practiced writing lexer rules for the simple Calc language, you should **start work on your coursework**. Please do ask for help from the tutors, but **do not expect answers to be given** – your coursework is a formal piece of work which is to be worked on all term – some of what you do not understand right now will become clear in future weeks.

In this task, start to write your lexer rules, referring to the Decaf language specification document on Moodle.

3.1 Download the **Decaf Lexer Starter Files** from this unit's Moodle area (week 2 – (Term 2) Compilers) and unzip them to a suitable location. When you have unzipped them, you should see the following files:



At this point you should recognise these files. Open the Decaf.g4 file: you will find that some rules have been started for you. The **ID** rule is not correct/complete according to the Decaf Reference Grammar. You should start your coursework by completing the **ID** rule.

To **test your rule**, you should use the supplied test files:

Within the **testdata** folder you will find subfolders, one of which is called **lexer**. Inside the lexer you will see a bunch of text files with names like **char1** and **id1**, where the **char** and **id** parts of the filename indicate which rules these files are testing i.e. to test your id rule, use the id1, id2, and id3. You are also encouraged to try your own tests to ensure that you have written your lexer to spec.

Write a loop which iterates over all tokens and prints the name of the token, then test your ID rule

3.2 At this point you should independently work on the rest of your lexer rules, using the appropriate test files. You will continue to work on these rules in next week's lab before moving on to parser rules.

