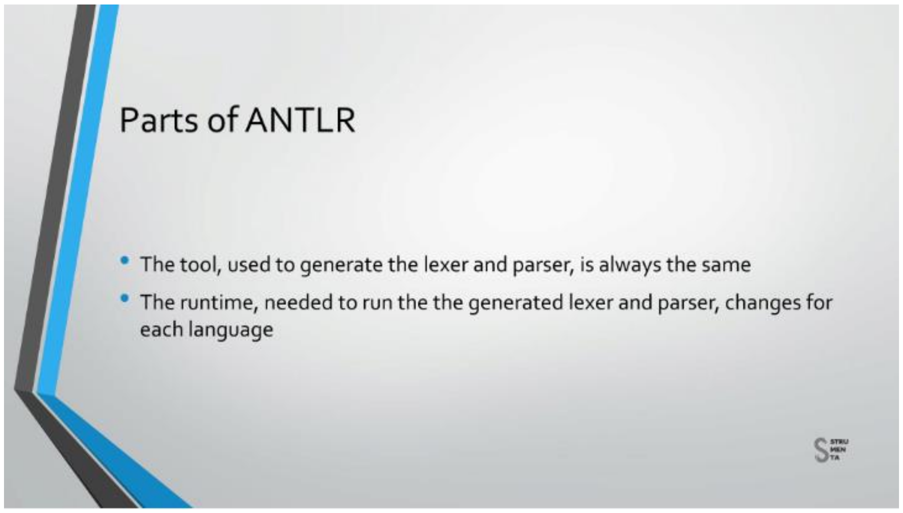
Lesson 02

Introduction to ANTLR

Hi, in this lesson we're going to explain how to set up ANTLR and how to use the common line tool.
We're also going to see what we can do with this, and how we can test the results that we produce.



Parts of ANTLR

- The tool, used to generate the lexer and parser, is always the same
- The runtime, needed to run the the generated lexer and parser, changes for each language

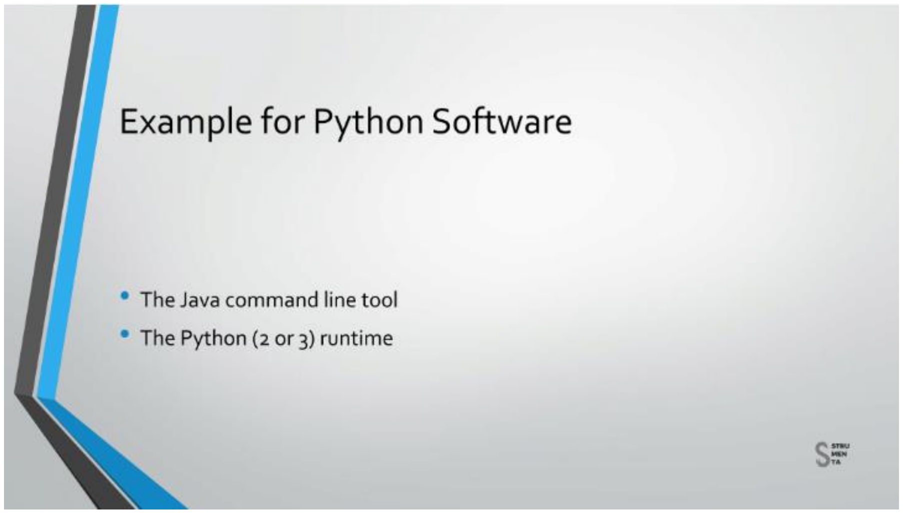
ANTLR is made by two different components: the first one is the tool, you will use it to generate a lexer and a parser for your language.
Then we have the runtime: this is a library that will be used in combination with the generated lexer and parser to process your language.

Only you will need the tool, while the runtime will need to be distributed together with your generated lexer and parser to your final users.

The tool is always the same, no matter which language you're targeting.
It's a Java program that you need on your development machine.

The runtime is different, there is one for each different target.

And they must be available both to the developer and to the user of the software.



Example for Python Software

- The Java command line tool
- The Python (2 or 3) runtime

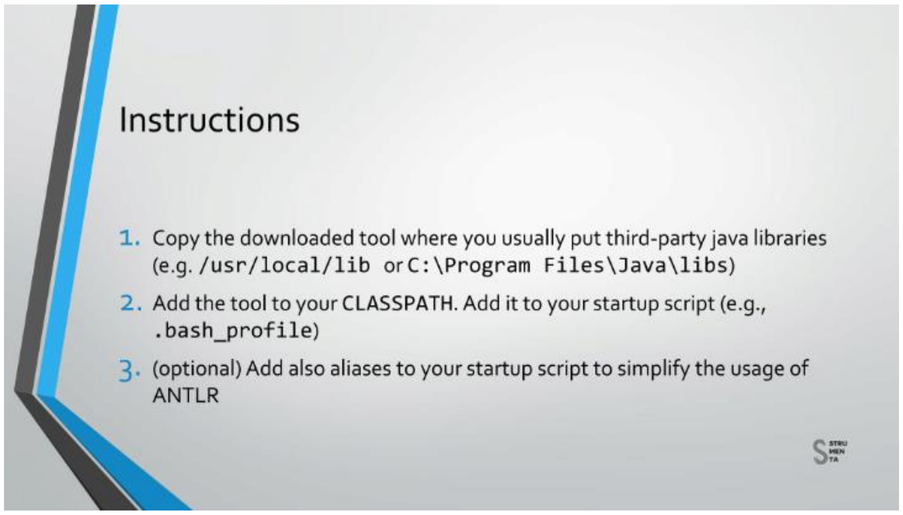
For instance if you were to generate and use ANTLR for Python, you're going to use the ANTLR tool to generate the parser in Python, and then you're going to use the Python runtime in order to run these generated parser.

The only requirement to use the tool, is to have Java 7 or higher installed on your machine.

Where to Find ANTLR

http://www.antlr.org/download.html

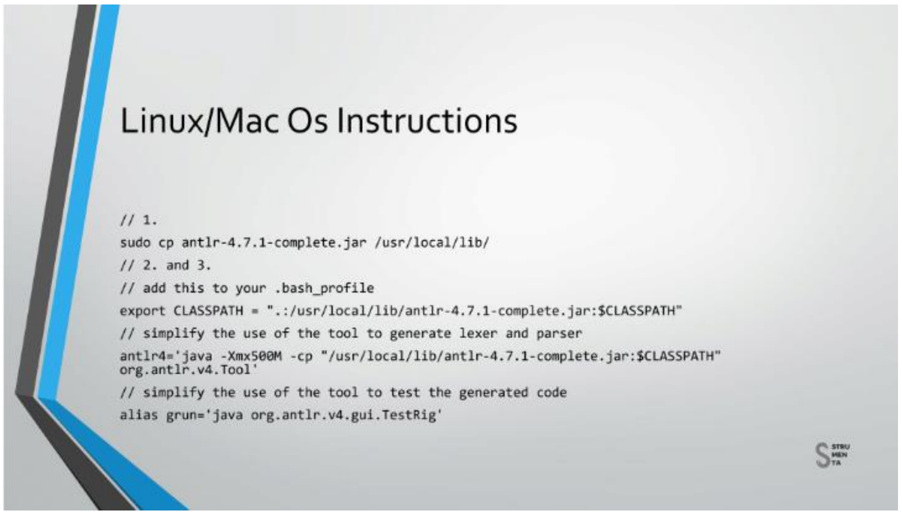
To install the tool you can simply download it from the official website.



Instructions

1. Copy the downloaded tool where you usually put third-party java libraries (e.g. `/usr/local/lib` or `C:\Program Files\Java\libs`)
2. Add the tool to your CLASSPATH. Add it to your startup script (e.g., `.bash_profile`)
3. (optional) Add also aliases to your startup script to simplify the usage of ANTLR

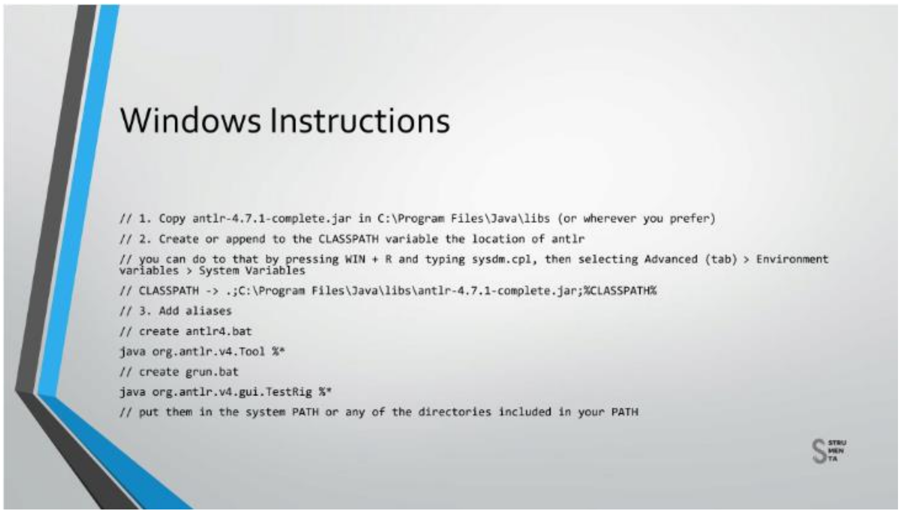
At this time the most recent version is version 4.7.1.



Linux/Mac Os Instructions

```
// 1.
sudo cp antlr-4.7.1-complete.jar /usr/local/lib/
// 2. and 3.
// add this to your .bash_profile
export CLASSPATH = ".:/usr/local/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
// simplify the use of the tool to generate lexer and parser
antlr4='java -Xmx500M -cp "/usr/local/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
org.antlr.v4.Tool'
// simplify the use of the tool to test the generated code
alias grun='java org.antlr.v4.gui.TestRig'
```

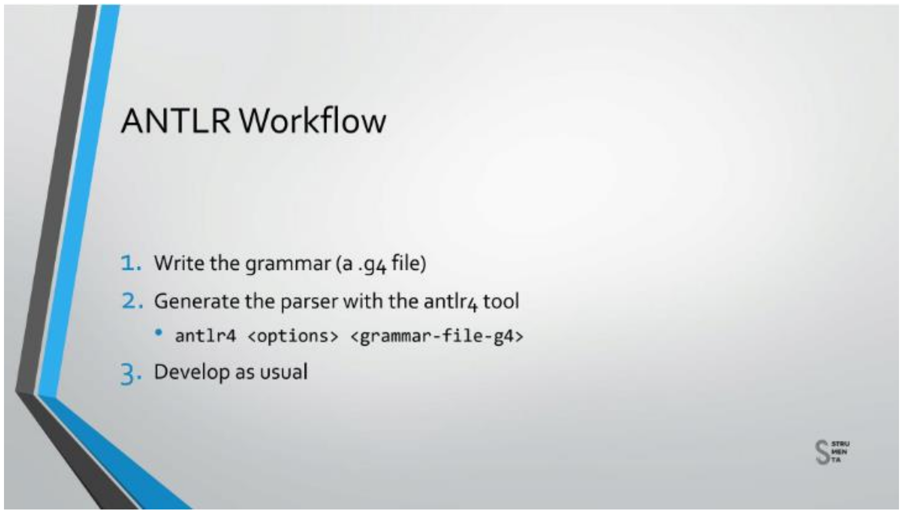
Once you've installed ANTLR, you need to make sure that it's available in your path, so that it can be easily used from the common line.



Windows Instructions

```
// 1. Copy antlr-4.7.1-complete.jar in C:\Program Files\Java\libs (or wherever you prefer)
// 2. Create or append to the CLASSPATH variable the location of antlr
// you can do to that by pressing WIN + R and typing sysdm.cpl, then selecting Advanced (tab) > Environment
variables > System Variables
// CLASSPATH -> .;C:\Program Files\Java\libs\antlr-4.7.1-complete.jar;%CLASSPATH%
// 3. Add aliases
// create antlr4.bat
java org.antlr.v4.Tool %*
// create grun.bat
java org.antlr.v4.gui.TestRig %*
// put them in the system PATH or any of the directories included in your PATH
```

Now, you're probably already familiar with this procedure, it varies depending on your operative system, but in any case we have reported all the comments you need to run in the slide.



ANTLR Workflow

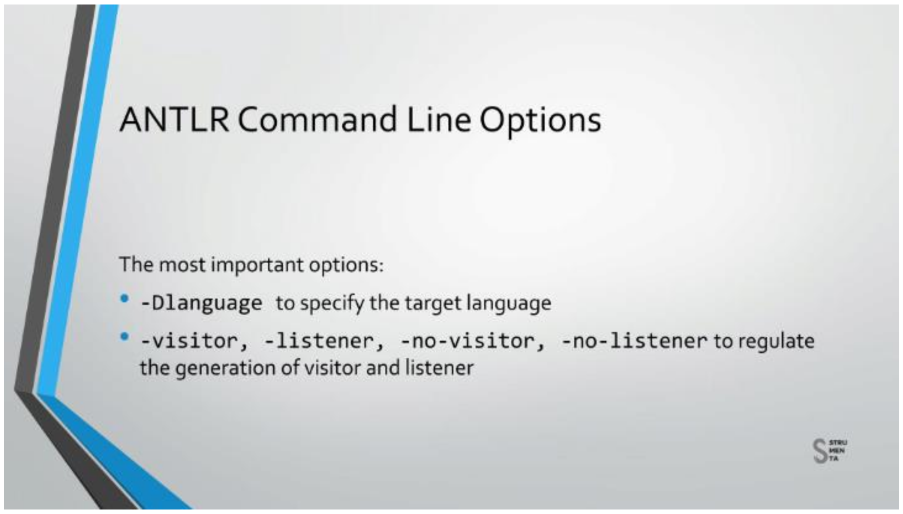
1. Write the grammar (a .g4 file)
2. Generate the parser with the antlr4 tool
   - `antlr4 <options> <grammar-file-g4>`
3. Develop as usual

The typical workflow to use ANTLR is not very complicated.

You start with a grammar file, that is a file with extension .g4, that contains the rules of your language.

Then you use the ANTLR tool to generate, from this grammar file, a lexer and a parser.

Finally you can execute the generated lexer and parser in combination with the runtime.



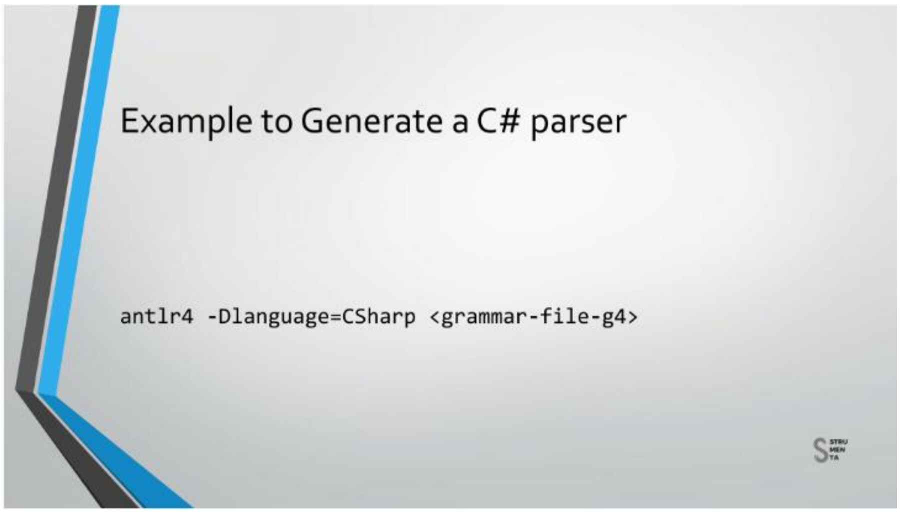
ANTLR Command Line Options

The most important options:

- `-Dlanguage` to specify the target language
- `-visitor, -listener, -no-visitor, -no-listener` to regulate the generation of visitor and listener

There are a couple of important options that you can use when running the ANTLR tool.
The first one is used to specify the target language: by default the generated lexer and parser will be Java program, but you can change that by using the specific option.
In this case you have just to specify which target language you want to use.

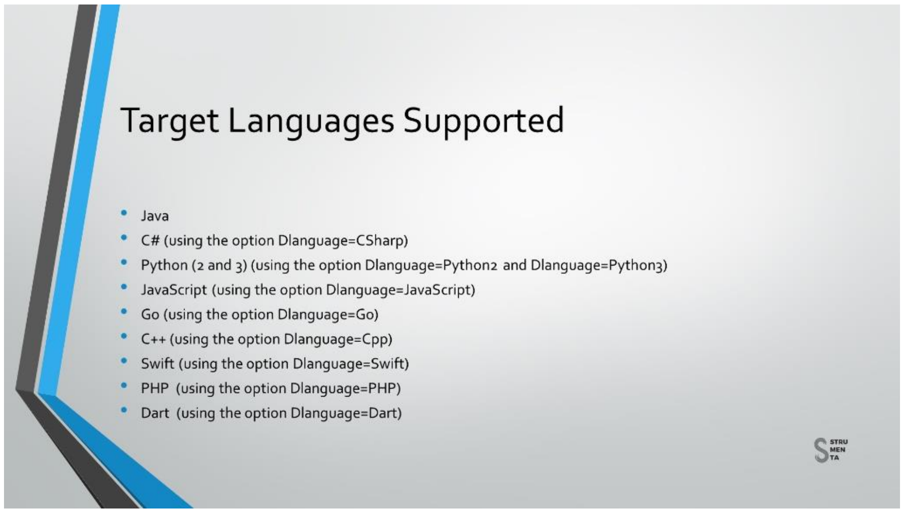
For instance, if you want to generate a C# parser, then you can use the instructions that are displayed in the slide.



Example to Generate a C# parser

```
antlr4 -Dlanguage=CSharp <grammar-file-g4>
```

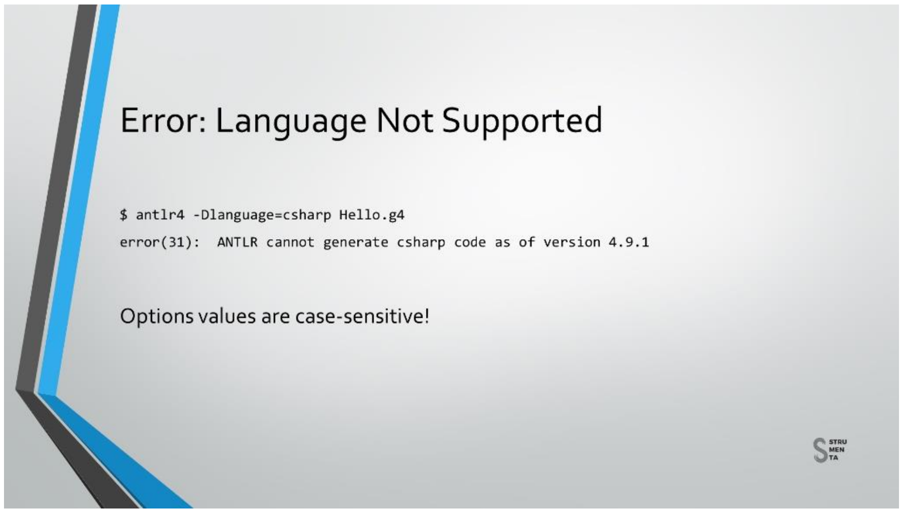
There are many targets that are supported by ANTLR.



Target Languages Supported

- Java
- C# (using the option Dlanguage=CSharp)
- Python (2 and 3) (using the option Dlanguage=Python2 and Dlanguage=Python3)
- JavaScript (using the option Dlanguage=JavaScript)
- Go (using the option Dlanguage=Go)
- C++ (using the option Dlanguage=Cpp)
- Swift (using the option Dlanguage=Swift)
- PHP (using the option Dlanguage=PHP)
- Dart (using the option Dlanguage=Dart)

Java which is the default one, then C#, Python 2 and Python 3, JavaScript, Go, C++, Swift.

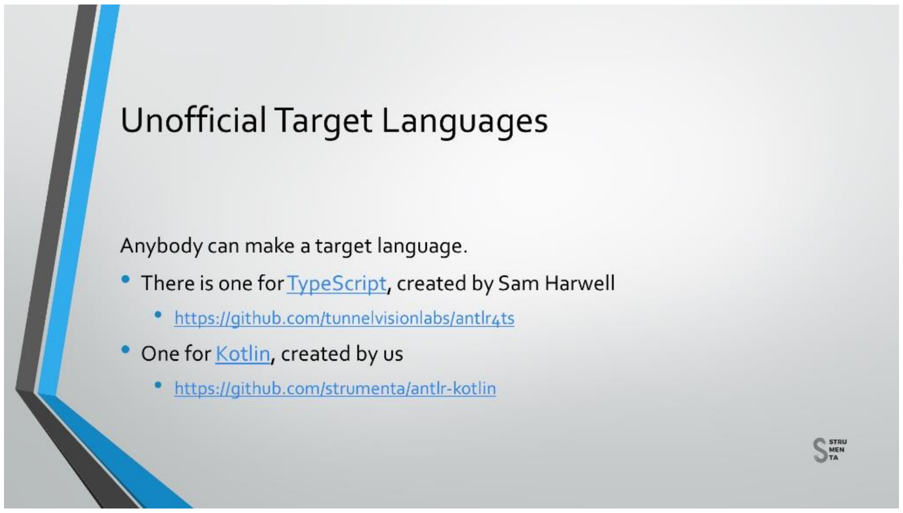


Error: Language Not Supported

```
$ antlr4 -Dlanguage=csharp Hello.g4
error(31):  ANTLR cannot generate csharp code as of version 4.9.1
```

Options values are case-sensitive!

Remember that the target language name is case sensitive. So, if you are using the wrong case, well, you will get an error. The message will say that the language is not supported, so pay attention to that.



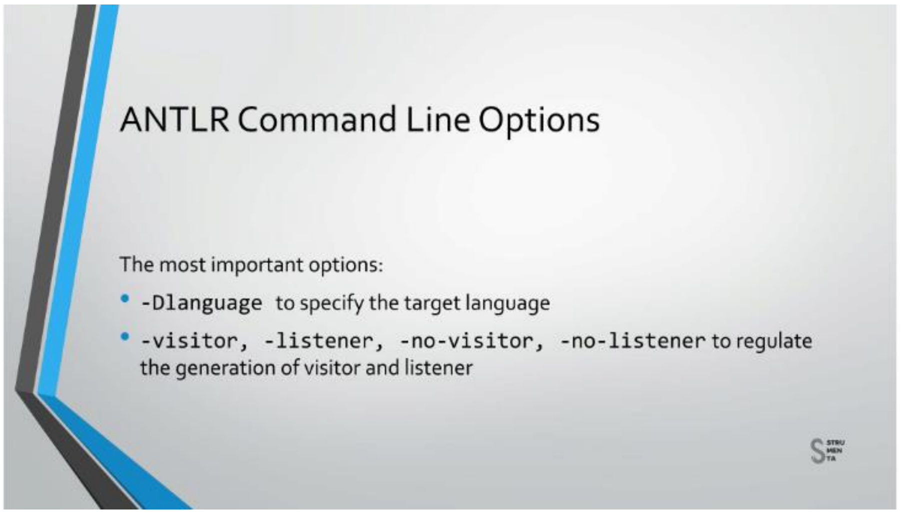
Unofficial Target Languages

Anybody can make a target language.

- There is one for TypeScript, created by Sam Harwell
  - https://github.com/tunnelvisionlabs/antlr4ts
- One for Kotlin, created by us
  - https://github.com/strumenta/antlr-kotlin

There are also a few unofficial target language that were created by other developers: for instance, there is one for TypeScript, that was created by Sam Harwell, and one for Kotlin, that well, we created.

These are experimental targets, so they could not have the same level of maturity than the official targets.

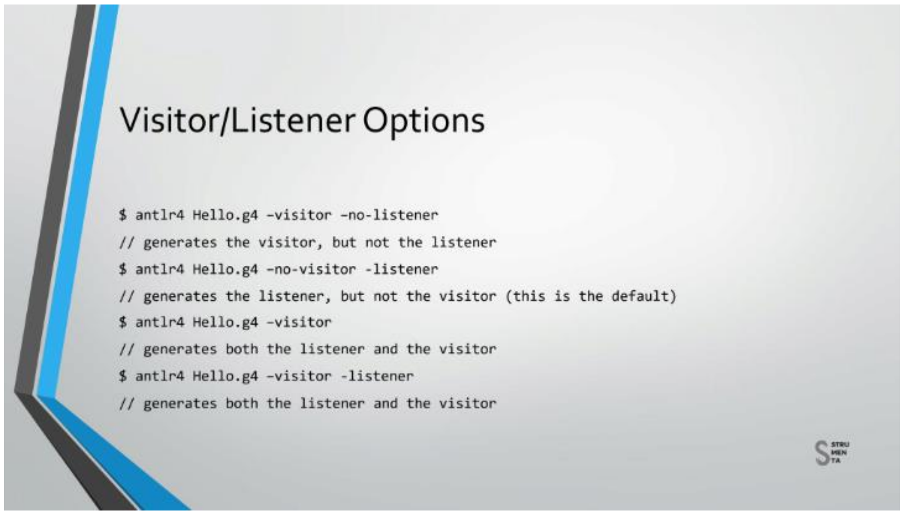


ANTLR Command Line Options

The most important options:

- `-Dlanguage` to specify the target language
- `-visitor, -listener, -no-visitor, -no-listener` to regulate the generation of visitor and listener

The other important options control if you're going to generate a visitor or a listener.
Visitors and listeners are useful tools to be used together with the parser.
Don't worry if you have no idea what they are, because we're going to explain that.

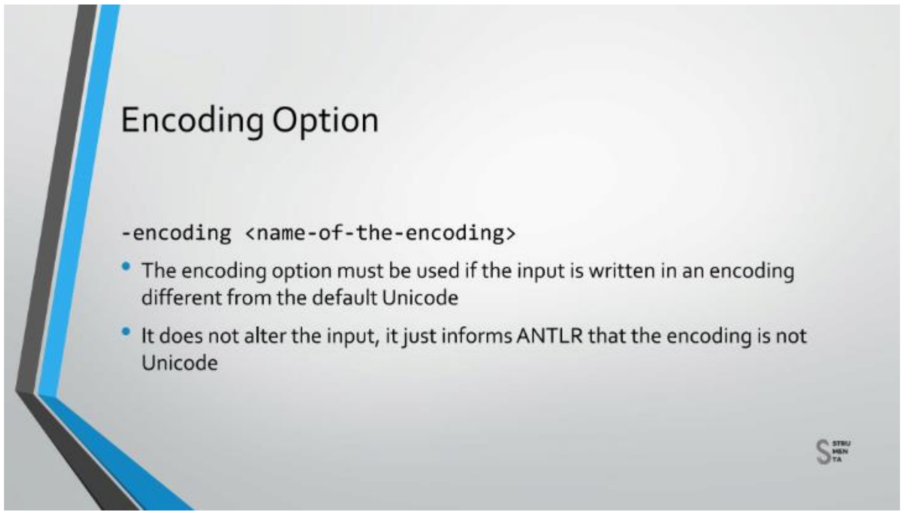


Visitor/Listener Options

```
$ antlr4 Hello.g4 -visitor -no-listener
// generates the visitor, but not the listener
$ antlr4 Hello.g4 -no-visitor -listener
// generates the listener, but not the visitor (this is the default)
$ antlr4 Hello.g4 -visitor
// generates both the listener and the visitor
$ antlr4 Hello.g4 -visitor -listener
// generates both the listener and the visitor
```

By default ANTLR generates only the listener, so to create the visitor you need to use the visitor option from the common line.
If instead you don't want to generate the listener, you use the no-listener option.
There also the opposite options no-visitor and listener, but these are the default options so you can omit them.

Remember that you can also generate both a visitor and a listener, you're not forced to pick one.



Encoding Option

```
-encoding <name-of-the-encoding>
```

- The encoding option must be used if the input is written in an encoding different from the default Unicode
- It does not alter the input, it just informs ANTLR that the encoding is not Unicode

There are also other options that you may want to use from time to time.

Imagine the deep encoding of your input is different from the default encoding that is UTF-8.

You could convert your input on-the-fly,, but this will also alter any output, which the user may not like.

So what you can do instead, is write in the grammar in the same encoding of the input.

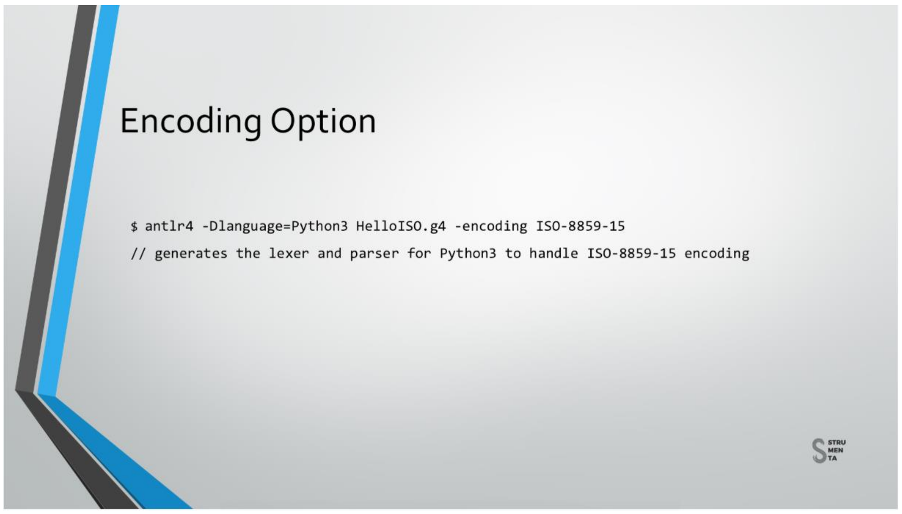
However if you do that, you must also tell ANTLR in which encoding you've wrote the grammar.

That's because ANTLR by default expects the grammar to be encoded in UTF-8, so the encoding option must be used if the grammar file is written and encoded in different from UTF-8.

As we said this option does not alter the encoding of the input to the parser.

You just inform ANTLR that the grammar file will be encoded in the specified encoding.

So you write the grammar in the coded you expect, and then you use the option to inform the parser, that the grammar uses that particular encoding.



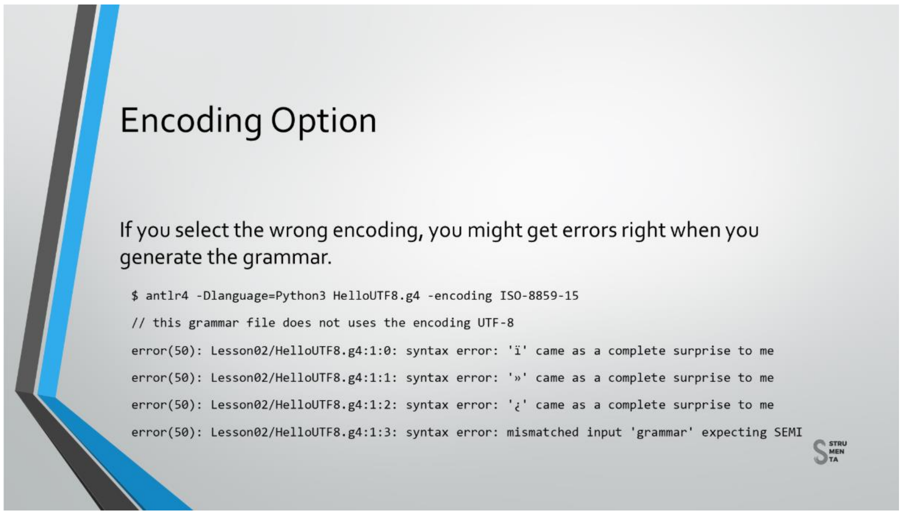
Encoding Option

```
$ antlr4 -Dlanguage=Python3 HelloISO.g4 -encoding ISO-8859-15
// generates the lexer and parser for Python3 to handle ISO-8859-15 encoding
```

For instance you can use this option to say that the grammar is written using the encoding IS0-8859-15.

If you set the wrong encoding then you're going to get all sorts of nasty errors when you try to generate the parser using ANTLR.

Let me repeat, it's not about the characters they use in the grammar, it is about the encoding in which the grammar file itself is written.

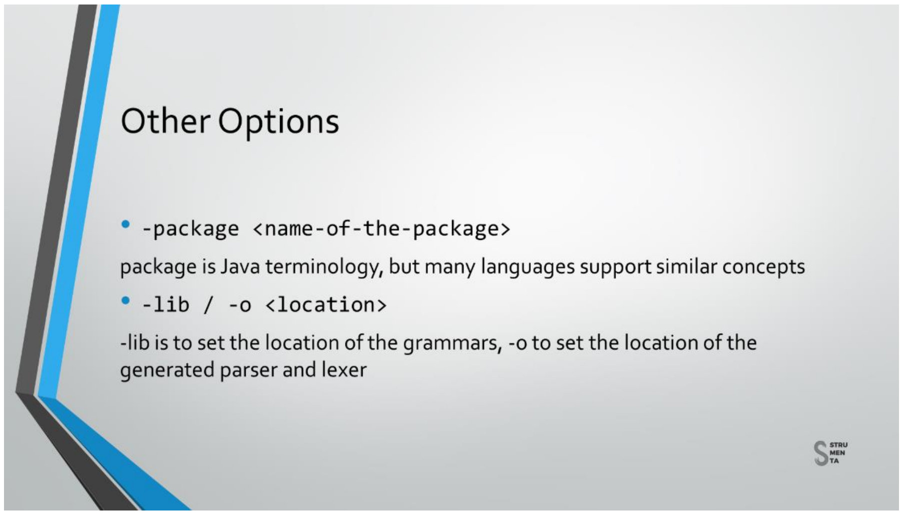


Encoding Option

If you select the wrong encoding, you might get errors right when you generate the grammar.

```
$ antlr4 -Dlanguage=Python3 HelloUTF8.g4 -encoding ISO-8859-15
// this grammar file does not uses the encoding UTF-8
error(50): Lesson02/HelloUTF8.g4:1:0: syntax error: 'ï' came as a complete surprise to me
error(50): Lesson02/HelloUTF8.g4:1:1: syntax error: '»' came as a complete surprise to me
error(50): Lesson02/HelloUTF8.g4:1:2: syntax error: '¿' came as a complete surprise to me
error(50): Lesson02/HelloUTF8.g4:1:3: syntax error: mismatched input 'grammar' expecting SEMI
```

For instance, in this example the grammar file was written using the UTF-8 encoding, but, we say that the encoding was ISO-8859-15. So we receive errors like these ones.



Other Options

- `-package <name-of-the-package>`

package is Java terminology, but many languages support similar concepts

- `-lib / -o <location>`

-lib is to set the location of the grammars, -o to set the location of the generated parser and lexer

The package option, adds a package decoration to the generated lexer and parser.

The concept of package is present in languages like Java, but other targets could have something roughly equivalent, like the C# slim spaces. Why some targets do not support these options at all?

These options and the concept of packages are useful to organize your code in different name spaces, modules, whatever you want to call them.

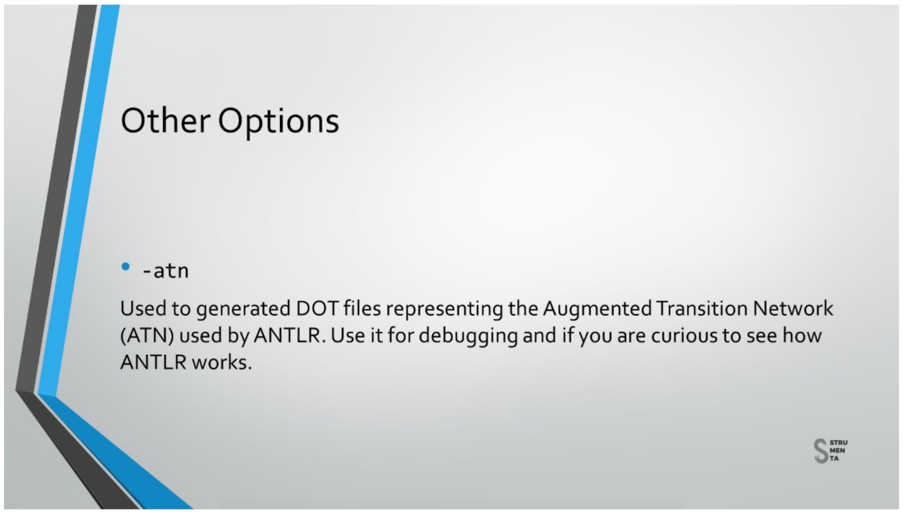
It's mostly useful when you are writing a larger program and you want integrate the lexer and the parser in something more complex and that is frequently the case.

The lib option is used to indicate the location of the grammars.

While the -o option is used to see where to put the generated code.

One case in which you want to use the lib option is when your grammar relies on a base grammar, that is present not in the core directory of your project but somewhere else.

You generally use your option when you need your generated code to be in a specific location. For example because your bill process expects these generated code to be in a certain directory, in order to later compile it.



Other Options

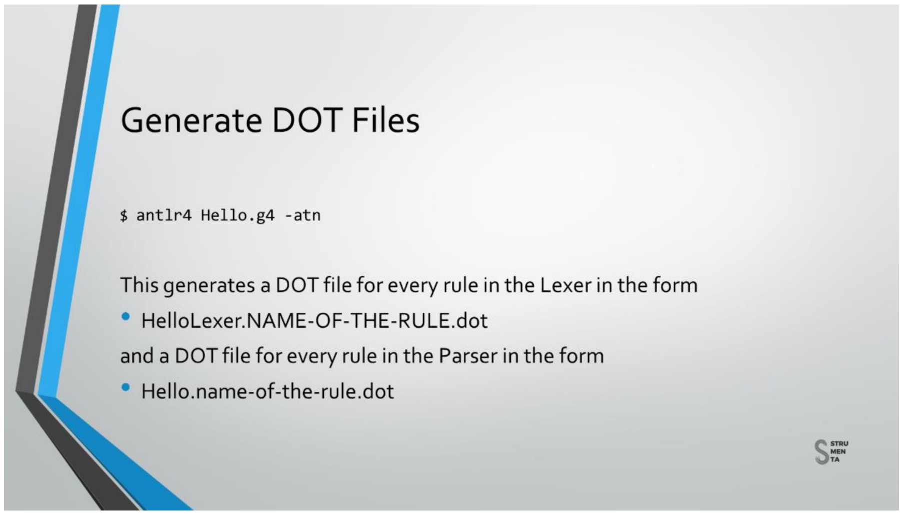
- `-atn`

Used to generated DOT files representing the Augmented Transition Network (ATN) used by ANTLR. Use it for debugging and if you are curious to see how ANTLR works.

Finally the ATN option is used to generate DOT files representing the ATN or Augmented Transition Network. This is an internal data structure that ANTLR uses to take decisions. In simple terms the ATN is a format that contains the information we provided in a grammar, in a way that is simpler to use for ANTLR.

This is an advanced option used for debugging. If your program misbehave or simply behaves differently from what you expect, you can use this option to take a look at the ATN that is used internally by ANTLR.

You can also be useful to understand how ANTLR works internally, if you're interested in that sort of things.



Generate DOT Files

```
$ antlr4 Hello.g4 -atn
```

This generates a DOT file for every rule in the Lexer in the form

- HelloLexer.NAME-OF-THE-RULE.dot

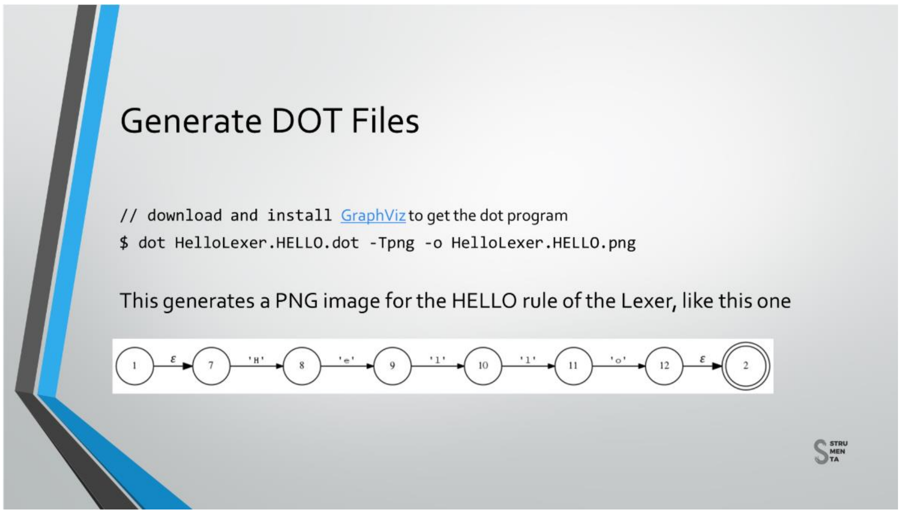
and a DOT file for every rule in the Parser in the form

- Hello.name-of-the-rule.dot

What happens in practice is that this option generates DOT files for each of the rules of your grammar.

The DOT format is a textual format used to represent graphs, so you will need a program to visualize that.

The program that you're going to use is GraphViz that includes the DOT utility, that will generate images corresponding to those DOT files.



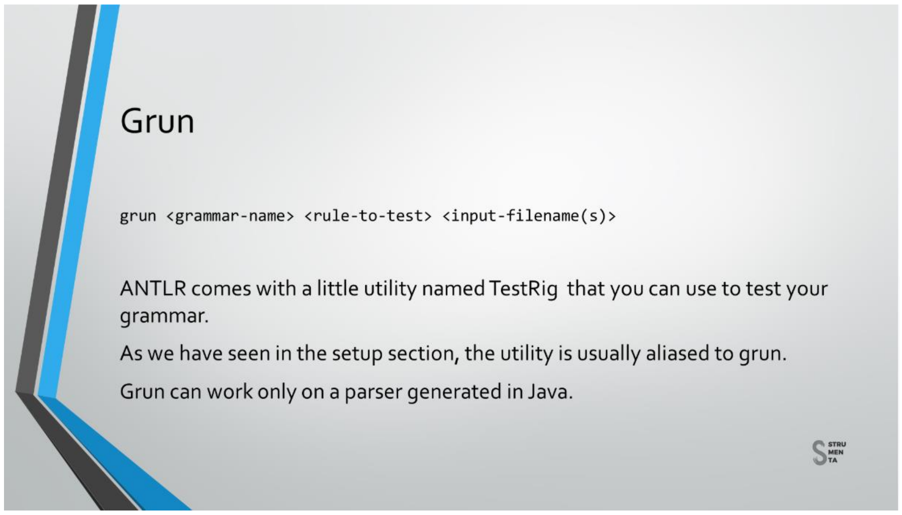
Generate DOT Files

```
// download and install GraphViz to get the dot program
$ dot HelloLexer.HELLO.dot -Tpng -o HelloLexer.HELLO.png
```

This generates a PNG image for the HELLO rule of the Lexer, like this one

1
ε
7
'H'
8
'e'
9
'l'
10
'l'
11
'o'
12
ε
2

As you can see in the rule 'Hello' in this example matches just the 'Hello' string.



Grun

```
grun <grammar-name> <rule-to-test> <input-filename(s)>
```

ANTLR comes with a little utility named TestRig that you can use to test your grammar.

As we have seen in the setup section, the utility is usually aliased to grun.

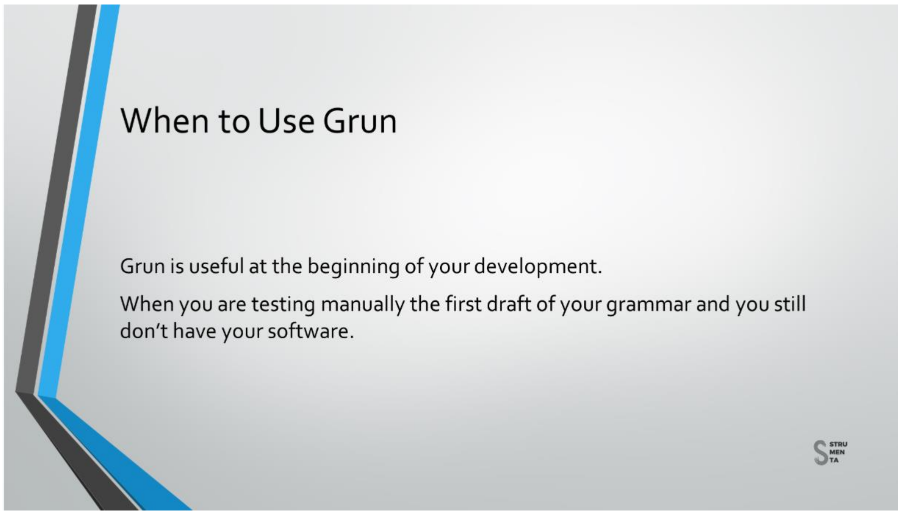
Grun can work only on a parser generated in Java.

You can also test your grammar using a little utility named TestRig.

As you've seen in the setup section the utility is usually aliased has Grun.

I'm not sure why but that's a tradition, so this is the name that you're going to use from now on.
To use Grun you need a grammar, a rule to test and an input.

Grun can read input from the common line or a file.
So the file name orfile names, are optional and you can instead analyze the input that you're going to provide directly in the console, or anyway from the standard input.
The testing tool is written in Java and can only work with Java parsers, so if you want to use it you need to generate a Java parser even if you plan to later use a different target. It's not a big deal, but you just remember that.
You have to generate the parser in Java and then you have to compile it.
Otherwise you're going to hear Grun complaining that it cannot find a grammar with that name.

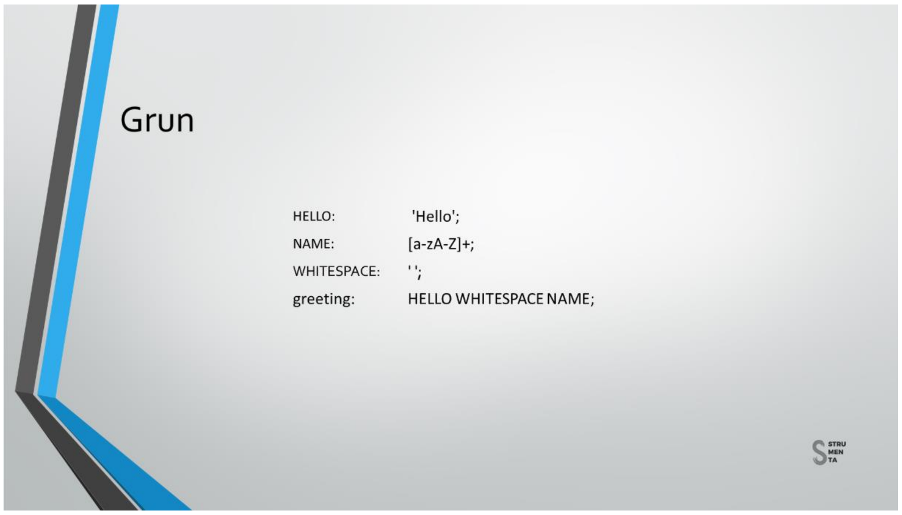


When to Use Grun

Grun is useful at the beginning of your development.

When you are testing manually the first draft of your grammar and you still don't have your software.

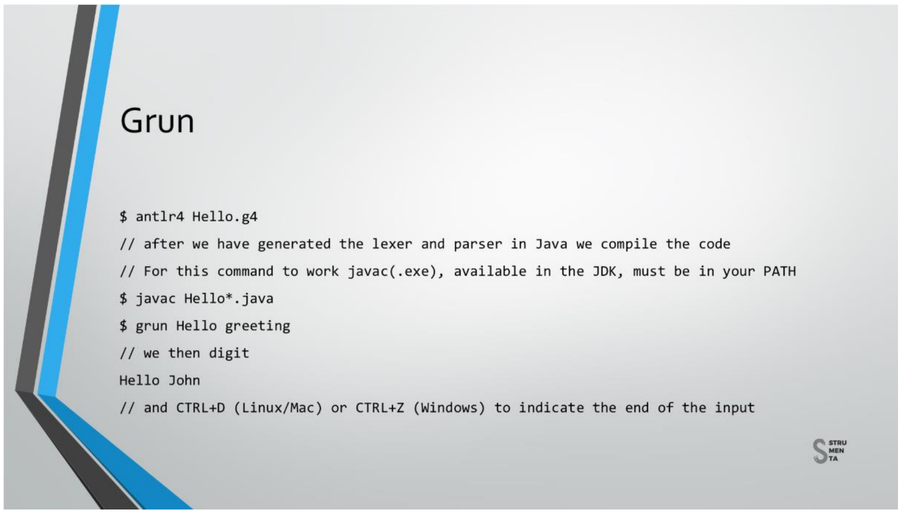
Grun is very useful at the beginning of your development when you're manually testing the first draft of your grammar and you still don't have created the rest of your software.
As your grammar becomes more stable, you may want to rely on automated tests.
In a later lesson we will see how to write such tests.



Grun

```
HELLO:          'Hello';
NAME:           [a-zA-Z]+;
WHITESPACE:     ' ';
greeting:       HELLO WHITESPACE NAME;
```

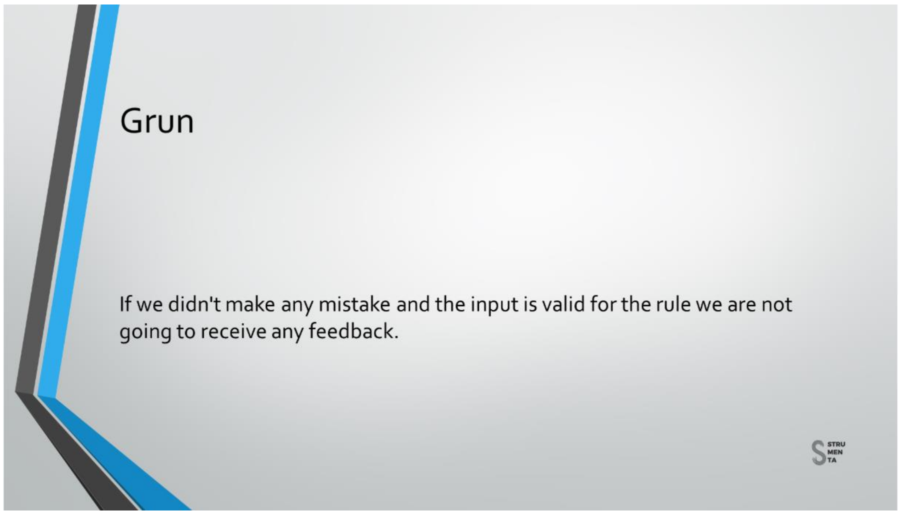
Let's see some example with the grammar 'Hello' that we have already seen in a previous lesson.



Grun

```
$ antlr4 Hello.g4
// after we have generated the lexer and parser in Java we compile the code
// For this command to work javac(.exe), available in the JDK, must be in your PATH
$ javac Hello*.java
$ grun Hello greeting
// we then digit
Hello John
// and CTRL+D (Linux/Mac) or CTRL+Z (Windows) to indicate the end of the input
```

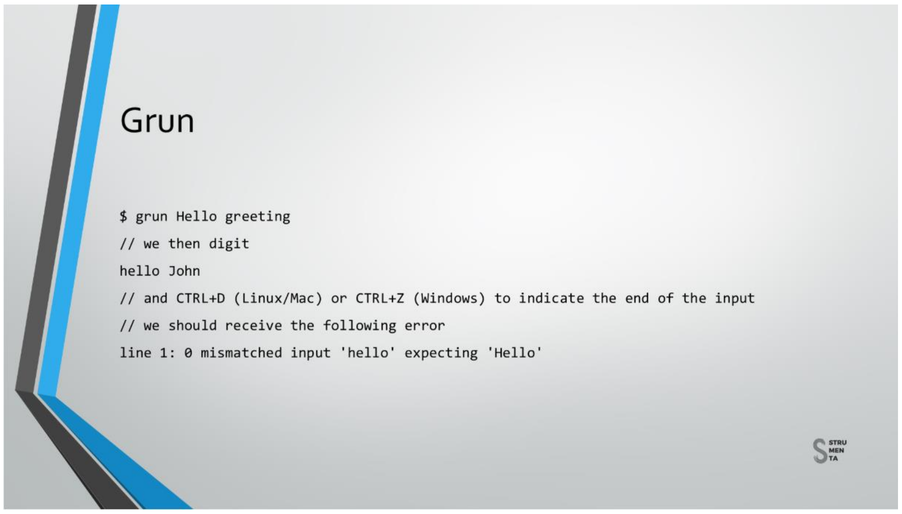
First we have to generate the lexer and parser with ANTLR4 and compile the resulting Java files. Then we can use Grun to check whatever the rule greeting works or not.



Grun

If we didn't make any mistake and the input is valid for the rule we are not going to receive any feedback.

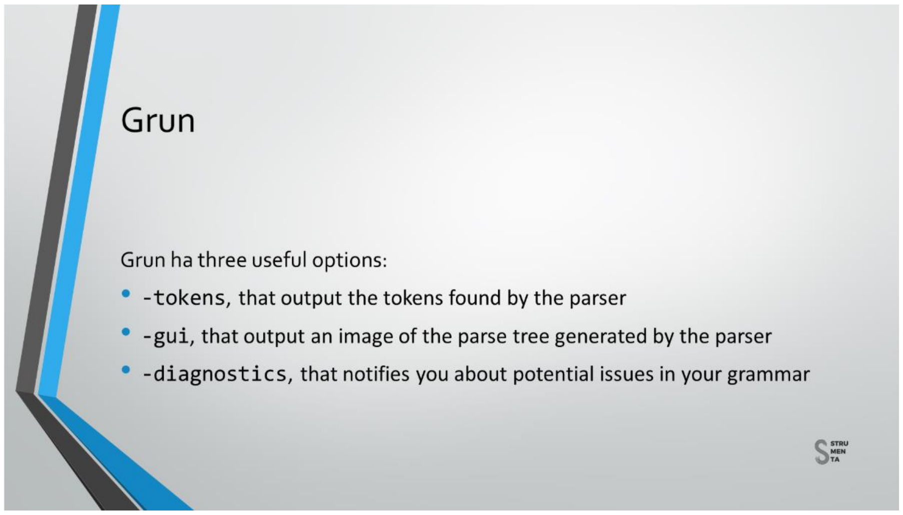
If we didn't commit any error, we are not going to receive any feedback, so silence is golden and indicates that the input is valid for the rule that you're testing.



Grun

```
$ grun Hello greeting
// we then digit
hello John
// and CTRL+D (Linux/Mac) or CTRL+Z (Windows) to indicate the end of the input
// we should receive the following error
line 1: 0 mismatched input 'hello' expecting 'Hello'
```

A wrong input will make Grun outputs some errors, for example, if you didn't capitalize 'H' in 'Hello'you're going to receive an error.



Grun

Grun ha three useful options:

- `-tokens`, that output the tokens found by the parser
- `-gui`, that output an image of the parse tree generated by the parser
- `-diagnostics`, that notifies you about potential issues in your grammar

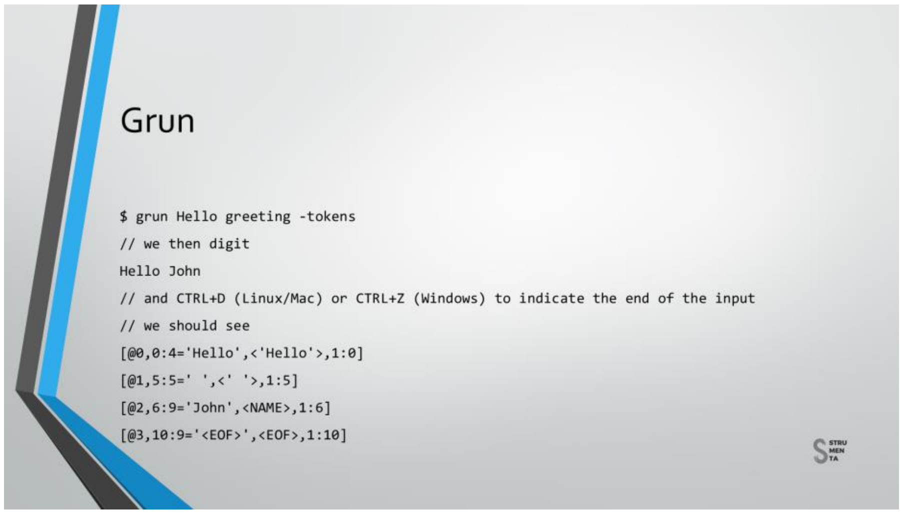
Grun also has a few useful options, we have -tokens, -gui and -diagnostics.

The option -tokens show the tokens detected by the parser;

-gui can be used to generate an image of the parse tree;

-diagnostics permits to get informed about potential issues with your grammar.

By using the option token, you can see all the tokens recognized by the parser.

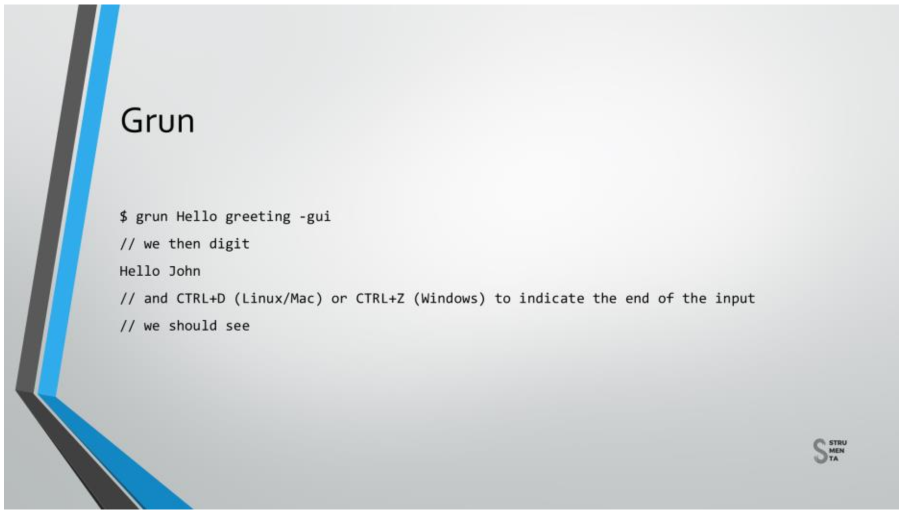


Grun

```
$ grun Hello greeting -tokens
// we then digit
Hello John
// and CTRL+D (Linux/Mac) or CTRL+Z (Windows) to indicate the end of the input
// we should see
[@0,0:4='Hello',<'Hello'>,1:0]
[@1,5:5=' ',<' '>,1:5]
[@2,6:9='John',<NAME>,1:6]
[@3,10:9='<EOF>',<EOF>,1:10]
```

These options also allow to see something interesting, you can see then the EOF token is automatically recognized even if it's not created in our grammar.
By the way EOF stand for end of file.

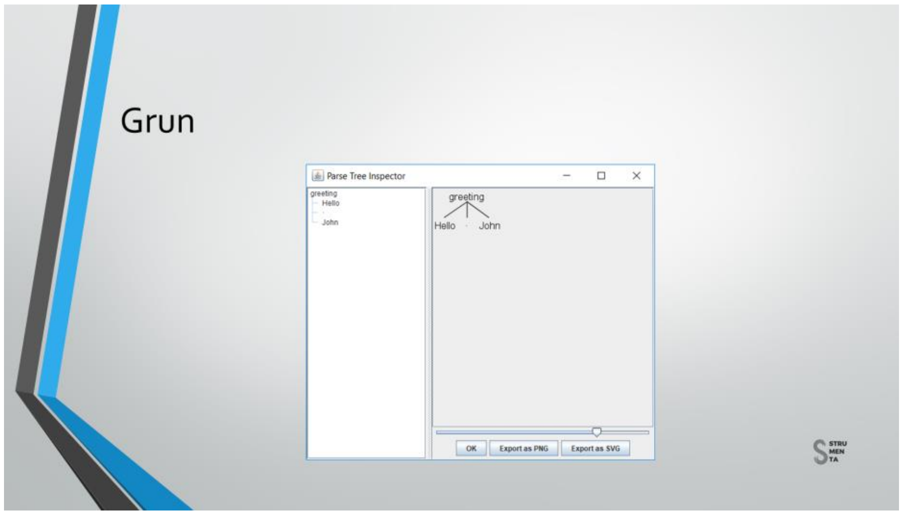
Using the option gui you can see a visual representation of the parse tree that corresponds to input we're precising.



Grun

```
$ grun Hello greeting -gui
// we then digit
Hello John
// and CTRL+D (Linux/Mac) or CTRL+Z (Windows) to indicate the end of the input
// we should see
```

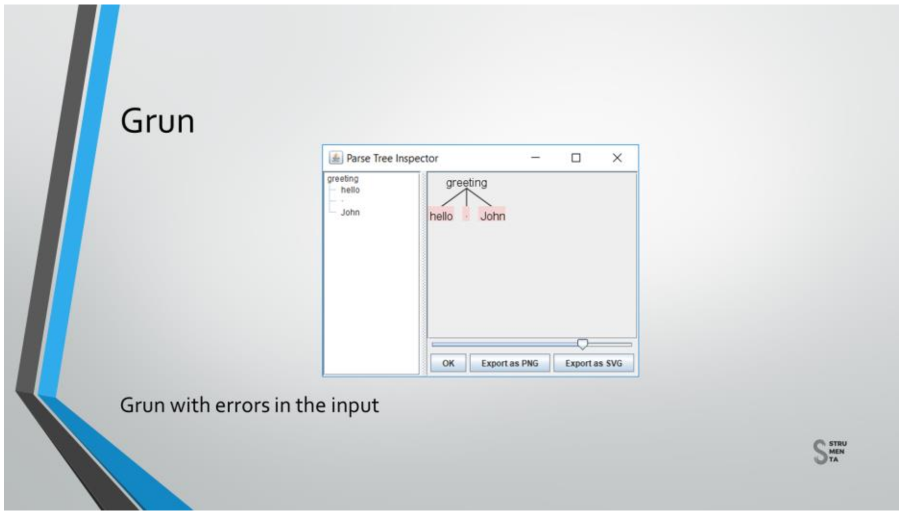
The gui option is the most useful when you're trying to see at a glance, how a problematic input is parsed.



This option is probably the only reason that you're going to keep using Grun later in the development stage.

If you use the token options with real input you're going to be probably overwhelmed by the amount of tokens that you are going to see.

Instead even with real data, the image created by the gui option is still very useful. Especially because it will also show if the input was not formed, so if the input was not completely matched by rule you're going to see errors in the image of the tree.

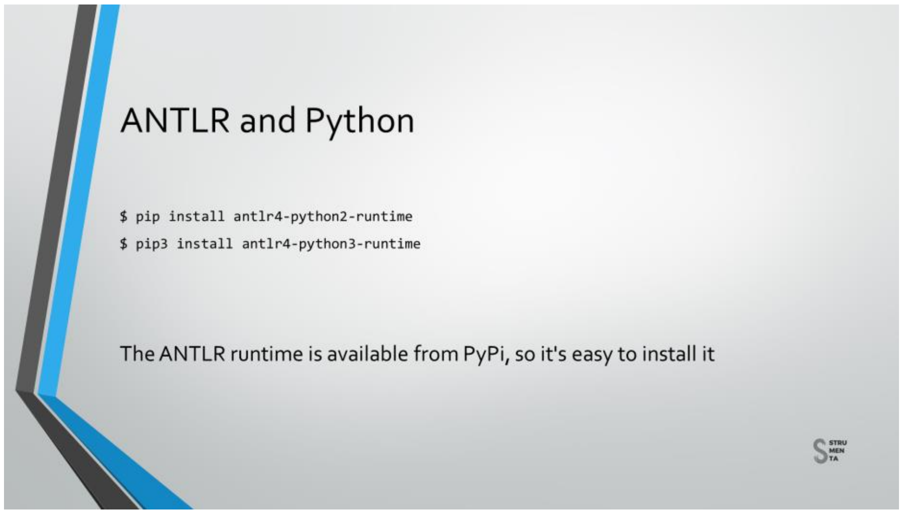


Grun

Grun with errors in the input

Now we can see how to setup a Python ANTLR program.

Generally you put the grammar files in the same folders as your Python source files.

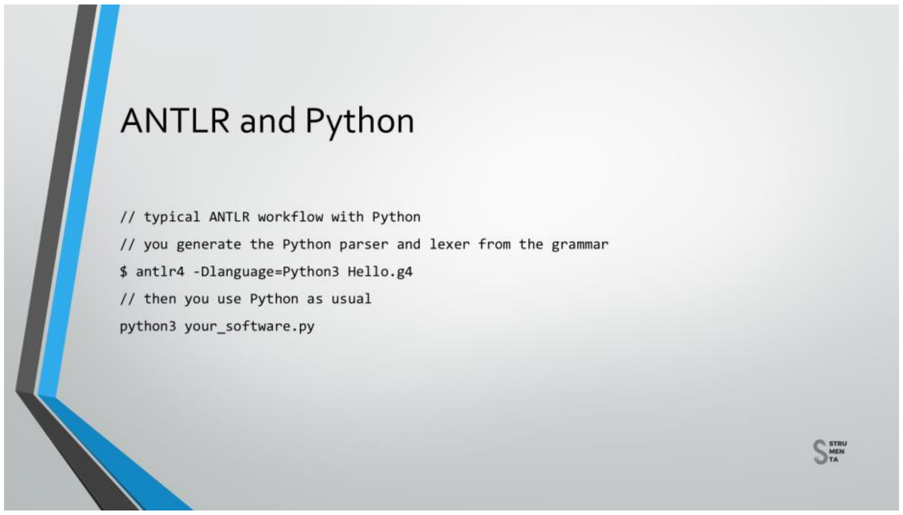


ANTLR and Python

```
$ pip install antlr4-python2-runtime
$ pip3 install antlr4-python3-runtime
```

The ANTLR runtime is available from PyPi, so it's easy to install it

If you do not like this layout, you can pick a different one and just use the lib and -o options that you've seen previously.
To create our Python parser we have to remember to use the correct option to specify the Python target.

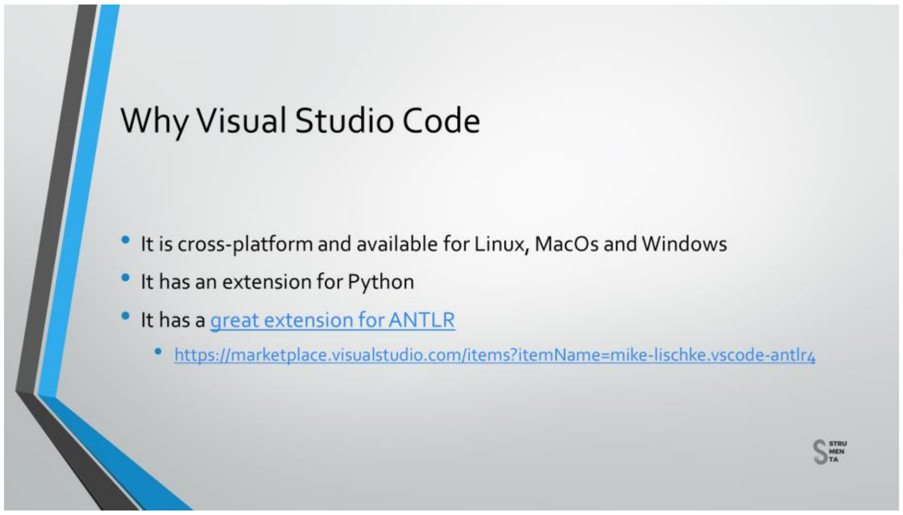


ANTLR and Python

```
// typical ANTLR workflow with Python
// you generate the Python parser and lexer from the grammar
$ antlr4 -Dlanguage=Python3 Hello.g4
// then you use Python as usual
python3 your_software.py
```

For Python you have to consider that we have two separate targets, one for Python2 and one for Python3. In a way they are treated like they were two separate languages and they also have two separate run times.
The run times are available on PyPi, that means you that you can install them using Pip.
You have to pick the matching runtime version for the target you are using, either Python2 or Python3.

There are many ideas that you could use with Python, in this course we are going to use Visual Studio Code.

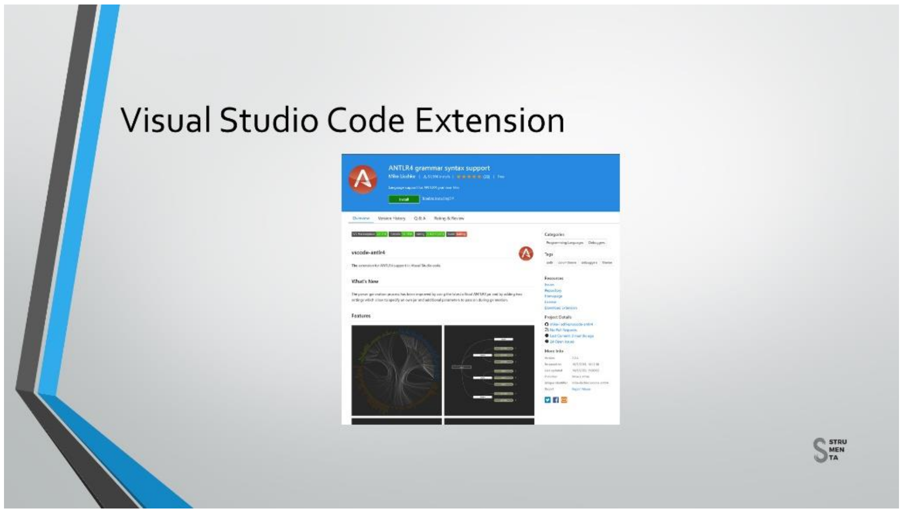


Why Visual Studio Code

- It is cross-platform and available for Linux, MacOs and Windows
- It has an extension for Python
- It has a great extension for ANTLR
    - https://marketplace.visualstudio.com/items?itemName=mike-lischke.vscode-antlr4

It's cross-platform first of all, so it's available on Linux, MacOs and Windows. It has an extension for Python and it's a very good extension for ANTLR.

The extensions offer the basic, so you get syntax highlighting and code completion.



Visual Studio Code Extension

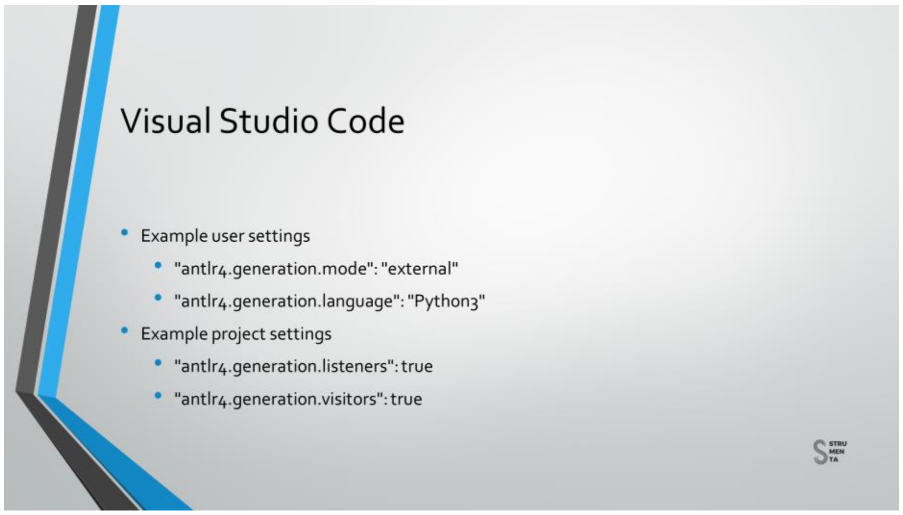
ANTLR4 grammar syntax support

But it can also do much more, it can code the ANTLR tool to automatically check for errors in your grammar and generate a ATM graph.

In addition to that, the extension can generate railroad diagrams. Railroad diagrams are not created by ANTLR itself but they're created by the extension. So you cannot use them to debug a parser generated by ANTLR.

That's because they don't tell you anything about the internal behavior of ANTLR, but you can use them to make sure you understand what a specific rule actually does.
If you need a graphical representation to be included in your documentation, well they're perfect.

Finally this extension can also help you with formatting the grammar.
The extension can generate the parser automatically, but by default the extension only generates the parser for internal use.



Visual Studio Code

- Example user settings
  - "antlr4.generation.mode": "external"
  - "antlr4.generation.language": "Python3"
- Example project settings
  - "antlr4.generation.listeners": true
  - "antlr4.generation.visitors": true

So it generates the parser internally to create railroad diagrams but you cannot use the generated parser yourself.

We can change that, and to do that we can go in preference, settings and add the corresponding options. In this way we can generate for example a Python3 parser each time we save the grammar. You could put the projects specific options in the same section, just make sure to select the tab workspace settings.
For instance you can set these options to automatically generate both the listener and the visitor for a certain project.
We can see ANTLR4.generation.listeners true, ANTLR4.generation.visitors true.
The true sections are functionally equivalent, the only difference is that one contains the settings for the specific project while the other contains settings for the specific user.

In this lesson we have learnt how to set up ANTLR, and how to use the main features of the common line tool, you've seen what you can do with it, its options and how to use the testing tool that comes with ANTLR.
We've also seen how to setup Visual Studio Code to use ANTLR in a productive way. Thank you for watching and see you at the next lesson.