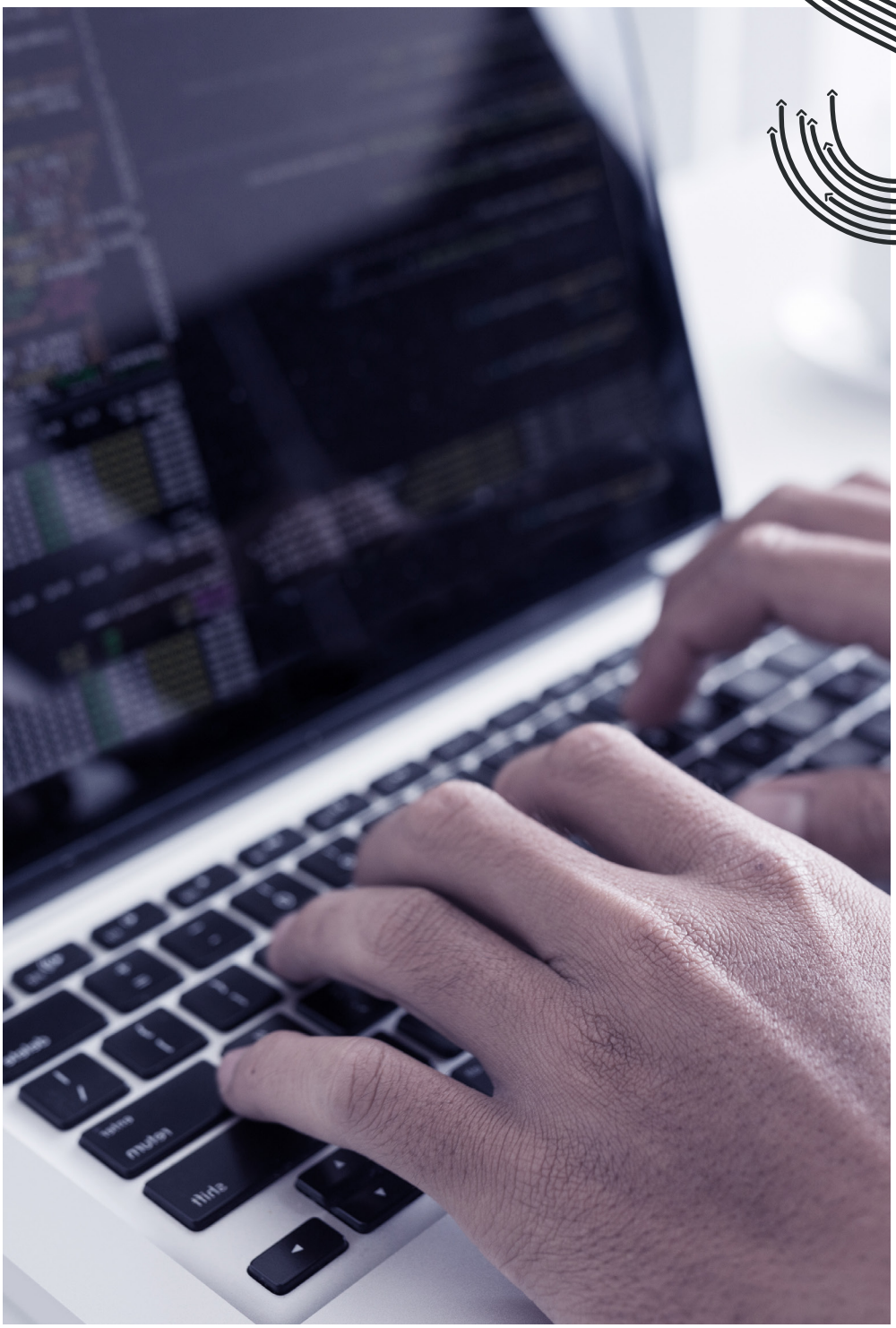
ANTLR
Cheatsheets



A Handy Reference
to make your life easier

STRUMENTA

Cheatsheets

This document contains cheatsheets for:

- Grammar
- Lexer
- Parser
- Actions and Clauses

Note that some cheatsheets contain a reference to the complete list of possible values for a specific element. For instance, in the lexer cheatsheet you can find the correct format for a lexer command, but the list of valid commands is too long to be placed in the cheatsheet so it is placed after the cheatsheet itself.

Grammar Cheatsheet

grammar Sum	The name of a grammar must be the same as the name of the file
lexer grammar Sum	A separate lexer grammar can be declared
parser grammar Sum	A separate parser grammar can be declared
import Sum	Other grammars can be imported
tokens { ONE }	Define tokens not associated to a rule
options {name=value;}	Set the value of an option (see Grammar Options for a list of valid options)
channels { ONE }	Create a new channel. Available only in lexer grammars
@header{ code }	Add the code to the header section of the grammar. In combined grammars it adds the code to both the lexer and the parser
@members{ code }	Add the code to the members section of the grammar. In combined grammars it adds the code to both the lexer and the parser
@lexer::header{ code }	Add the code to the header section of the lexer
@lexer::members{ code }	Add the code to the members section of the lexer
@parser::header{ code }	Add the code to the header section of the parser
@parser::members{ code }	Add the code to the members section of the parser
// Comment	A single line comment
/* Comment */	A Multi-line comment
'literal'	String and character literals
"literal"	Double quotes cannot be used as delimiters
'\u0001'	A literal character referenced by its own unicode code point (up to U+FFFF)
'\u{1F000}'	A literal character referenced by its own unicode code point (up to U+10FFFF)
'\n' '\r' '\b' '\t' '\f'	Escape sequences supported in string literals or characters sets

Grammar Options

There are two ways to input options relative to a grammar:

1. Using the -Dname command argument format
2. Putting them inside the **options** section of the grammar itself.

There is no difference in the results, but usually you prefer to use the second way, apart when setting the option language. That is because usually, when you set any of the other options, the grammar is fully dependent on some external code. In this case you want everybody that reads the grammar to know that. Instead you might use the option language on a language independent grammar.

The options are:

- language
- superClass
- TokenLabelType
- tokenVocab

language

```
options {language=CSharp;}
```

```
antlr4 -Dlanguage=CSharp Sum.g4
```

If the language is supported it generates the components (e.g., lexer, parser, etc.) in the specified target language. Remember to get the case of the language right.

The supported languages are:

Language	**Option**
Java	Java
C#	CSharp
Python 2	Python2
Python 3	Python3
JavaScript	JavaScript
Go	Go
C++	Cpp
Swift	Swift
PHP	PHP
Dart	Dart

superClass

```
options {superClass=SumSuper;}
```

Set the base class of the generated lexer (if used in *lexer grammars*) or parser (if used in *parser grammars or combined grammars).*

```
antlr4 -DsuperClass=SumSuper Sum.g4
```

When set on the command line, the option sets the base class for both the lexer and the parser.

When properly set, the declaration of the generated lexer, with the Python 3 target, should look like the following.

```
class SumLexer(SumSuper):
```

When properly set, the declaration of the generated parser lexer, with the Python 3 target, should look like the following.

```
class SumParser(SumSuper):
```

Usually it makes most sense to use two different base classes for the lexer and the parser. Each of the two should inherit from the respective base class for the component.

```
// super class for the lexer
class SumSuperLexer(Lexer):
// super class for the parser
class SumSuperParser(Parser):
```

TokenLabelType

```
options {TokenLabelType=SumToken;}
```

This option starts with an uppercase letter. ANTLR by default generates tokens with the **Token** class. It also adds fields of the proper class **Token** in the Context objects used in the parser, listener and visitor. You can change that by using this option.

You also need to create a custom **TokenFactory** class and set it to be used by the token stream used by the parser.

```
// custom token factory, which will also generate SumToken(s)
class CustomTokenFactory(TokenFactory):
// then, when setting up the token stream
tokens = CommonTokenStream(lexer)
tokens.tokenSource._factory = CustomTokenFactory()
```

For some target languages you might also need to add an import to the generated parser.

```
// inside the grammar
@parser::header {
from SumToken import SumToken
}
```

That is because they use it for the type of the **End of File** token.

```
// inside the generated SumParser
EOF = SumToken.EOF
```

tokenVocab

```
options {tokenVocab=SumLexerGrammar;}
```

By default, ANTLR assigns a number automatically to each token it encounters in the grammar. However, it cannot do that automatically in a *parser grammar*. In that case you need to use the tokens defined in a separate lexer grammar.
Basically, what this option does is reading the .tokens file corresponding to the lexer grammar.

Lexer Cheatsheet

rule:	Matches
TOKEN	Another token with name TOKEN. Recursion is allowed, but not left recursion
'literal'	String 'literal'
.	Any character
~'a'	Any character but the one in the set
[a-z]	Any character in the characters set. Inside characters set], \, and – must be escaped with \
[\p{Property}]	Any character matching the property. Only usable inside character sets (see Unicode Properties for a list of valid properties)
[\p{Property=value}]	Any character having property with the indicated value. Only usable inside character sets (see Unicode Properties for a list of valid properties)
'a'..'z'	Any character in the range
(A | B)	The token A or the token B
(A | B)?	Nothing or the token A or the token B
(A | B)*	Nothing or any number of token a or the token b
(A | B)+	At least one time the token a or the token b
(A | B)?? C	Nothing or the token A or the token B, the fewer that still allows to then match C
(A | B)*? C	Nothing or any number of token a or the token B, the fewer that still allows to then match C
(A | B)+? C	At least one time the token A or the token B, but the fewer that still allows to then match token C
Format	Element
PLUS : '+';	The name of a lexer rule must start with an uppercase letter
{action}	Execute the action code after matching the action
{semantic}?	Execute the semantic predicate code: if it evaluates to true it considers the rule, otherwise it does not
-> command	Execute the specific command (see Lexer Commands for a list of valid commands.

Unicode Properties Supported

In characters sets ANTLR supports the standard Unicode properties set in the official documentation Annex 44. You should look the official documentation to get the full list, a description and a list of associated Unicode code points.

Property names are case-insensitive, _ and - are treated identically.
ANTLR supports referencing:

- all code points associated to a property and
- all code points with a specific value of a property

These are two examples of their usage.

```
MATH    : [\p{Math}] ;
CYRILLIC : [\p{Script=Cyrillic}] ;
```

If you input a non-existing property the ANTLR command line tool will throw an error and will fail to generate the lexer, parser, etc. For example, trying this property.

```
FAKE    : [\p{FakeProperty}] ;
```

Will results in this exception.

```
java.lang.RuntimeException: set is empty
```

In addition to these standard properties, ANTLR also supports extra properties all related to emoji.

Extended_Pictographic and Emoji

Used as \p{Extended_Pictographic} and \p{Emoji}.

These are similar properties, but there is a difference. Emoji are the standard set of graphical representation of feelings, attitude, etc. that everybody knows, such as 😀 (GRINNING FACE). However, there are also simplified graphical representation of object, like the ⛨ (BLACK CROSS ON SHIELD). Think about Unicode point with the property extended pictographic as the kind of stuff that appears in traffic signs and the like. Some code points can belong to both categories.

EmojiPresentation

The EmojiPresentation property refer to the style of the emoji shown to the user. The *Emoji* style represent the classical style that you see in Smartphone. *Text* style instead is a more simplified representation, in black and white, that you can encounter in simpler programs.

ANTLR supports the value for this property

- \p{EmojiPresentation=EmojiDefault} Unicode code points that have an Emoji style by default, but also have a Text style
- \p{EmojiPresentation=TextDefault} Unicode code points that have a Text style by default, but also have an Emoji style
- \p{EmojiPresentation=Text} Unicode code points that only have a Text style

Lexer Commands

There are a few useful commands available to use in lexer rules. If there is more than one command, they must be separated with a comma.

The commands are:

- channel(CHANNEL_NAME)
- mode(MODE_NAME)
- popMode, pushMode(MODE_NAME)
- more
- skip
- type(TOKEN_NAME)

channel(CHANNEL_NAME)

```
COMMENT : '/*' .*? '*/' -> channel(HIDDEN);
```

Use channel to change the channel in which the token is sent. Channels are like production lines and allow to lookup independently different sets of tokens. By default there are two channels:

- DEFAULT_CHANNEL
- HIDDEN

The HIDDEN channel is not used by the parser to match rules, so it is well suited to receive comments and other content that does not matter to parsing, but it might be useful to keep around.

mode(MODE_NAME)

```
lexer grammar Tag;
START      : '<' -> more, mode(TAG) ;
WHITESPACE : [ \r\n]+ -> skip ;

mode TAG;
// everything ends up here
STRING     : '>' -> mode(DEFAULT_MODE) ;
TEXT       : . -> more ;
```

After matching the token, it moves to the mode used as argument of the command. The lexer will then only look at token of that mode. By default, all tokens are considered in the DEFAULT_MODE.

popMode, pushMode(MODE_NAME)

```
lexer grammar Tag;
START      : '<' -> more, pushMode(TAG) ;
WHITESPACE : [ \r\n]+ -> skip ;

mode TAG;
// everything ends up here
STRING     : '>' -> popMode ;
TEXT       : . -> more ;
```

The command pushMode works just like mode: it changes the mode to the one indicated. However, it then also pushes the old mode into a stack of modes. It is used together with popMode, that switches the mode to the last one added to the stack.

The two commands can be used to safely move between a series of modes that can be entered and exited more ways than one. For instance, imagine that you can access MODE_1 from MODE_2 and MODE_3, so when you exit MODE_1 you do not know where to go back to unless you record from which mode you entered.

more

```
lexer grammar Tag;
START      : '<' -> more, mode(TAG) ;
WHITESPACE : [ \r\n]+ -> skip ;

mode TAG;
// everything ends up here
STRING     : '>' -> mode(DEFAULT_MODE) ;
TEXT       : . -> more ;
```

The command more matches the rule but then continue looking for the next token. The next token will include the text matched by this rule.

It is used most often with lexical modes to match the delimiters together with the content that they match. In this example everything ends up as matched by the **STRING** token.

skip

```
WHITESPACE : [ \r\n]+ -> skip ;
```

This command is used to make the lexer match the token but then not sent it to the parser. Effectively this command makes the token invisible to the parser.

type

```
LONG_COMMENT : '/*' .*? '*/' -> type(COMMENT);
```

Command used to change the type of the token matched by the rule. It is particularly useful when using lexical modes to support string interpolation. Using this command, you can avoid repetitions in the parser.

Parser Cheatsheet

rule:	Matches
TOKEN	A token with name **TOKEN**
~TOKEN	Any token but the one with the name **TOKEN**
'literal'	String **'literal'** (is also a token)
rule1	Another parser rule with name **rule1**
.	Any token
(a | b)	The rule **a** or the rule **b**
(a | b)?	Nothing or the rule **a** or the rule **b**
(a | b)*	Nothing or any number of rule **a** or the rule **b**
(a | b)+	At least one time the rule **a** or the rule **b**
(a | b)?? c	Nothing or the rule **a** or the rule **b**, the fewer that still allows to then match **c**
(a | b)*? c	Nothing or any number of rule **a** or the rule **b**, the fewer that still allows to then match **c**
(a | b)+? c	At least one time the rule **a** or the rule **b**, but the fewer that still allows to then match **c**
Format	**Element**
speak:'say' NO;	The name of a parser rule must start with a lowercase letter
{action}	Execute the action code after matching the rue
{semantic}?	Execute the semantic predicate code: if it evaluates to true it considers the rule, otherwise it does not
| rule1 #label1 | rule2 #label2	Each alternative of the rule has a label. Label1Context and Label2Context classes and respective listener and visitor methods will be created for each label
| rule1 #label1 | rule2 #label1	The same labels can be repeated
~~| rule1 #label1~~ ~~| rule2~~	If one alternative has a label, all of the must have one

Actions and Clauses Cheatsheet

```
rule returns[variable_name] locals[variable_name]
@init { /* action code here */ }
@after { /* action code here */ }
: /* rule definition */ { /* action code here */  };
catch[RecognitionException e] { /* action code here */ }
finally { /* action code here */ }
```