



 [antlr](#) / [antlr4](#)<> **Code** Issues 566 Pull requests 67 Discussions Actions I master ▾

...

[antlr4](#) / [doc](#) / **lexer-rules.md****parrt** Merge branch 'master' into patch-1 ❌**History** 7 contributors

Raw

Blame



310 lines (232 sloc) | 13 KB

Lexer Rules

A lexer grammar is composed of lexer rules, optionally broken into multiple modes. Lexical modes allow us to split a single lexer grammar into multiple sublexers. The lexer can only return tokens matched by rules from the current mode.

Lexer rules specify token definitions and more or less follow the syntax of parser rules except that lexer rules cannot have arguments, return values, or local variables. Lexer rule names must begin with an uppercase letter, which distinguishes them from parser rule names:

```
/** Optional document comment */  
TokenName : alternative1 | ... | alternativeN ;
```

You can also define rules that are not tokens but rather aid in the recognition of tokens. These fragment rules do not result in tokens visible to the parser:

```
fragment  
HelperTokenRule : alternative1 | ... | alternativeN ;
```

For example, `DIGIT` is a pretty common fragment rule:

```

INT : DIGIT+ ; // references the DIGIT helper rule
fragment DIGIT : [0-9] ; // not a token by itself

```

Lexical Modes

Modes allow you to group lexical rules by context, such as inside and outside of XML tags. It's like having multiple sublexers, one for each context. The lexer can only return tokens matched by entering a rule in the current mode. Lexers start out in the so-called default mode. All rules are considered to be within the default mode unless you specify a mode command. Modes are not allowed within combined grammars, just lexer grammars. (See grammar `XMLLexer` from [Tokenizing XML](#).)

```

rules in default mode
...
mode MODE1;
rules in MODE1
...
mode MODEN;
rules in MODEN
...

```

Lexer Rule Elements

Lexer rules allow two constructs that are unavailable to parser rules: the `..` range operator and the character set notation enclosed in square brackets, `[characters]`. Don't confuse character sets with arguments to parser rules. `[characters]` only means character set in a lexer. Here's a summary of all lexer rule elements:

Syntax	
T	Match token T at the current input position. Tokens always begin with
'literal'	Match that character or sequence of characters. E.g., 'while' or '='.
[char set]	<p>Match one of the characters specified in the character set. Interpret <code>x</code> characters are interpreted as single special characters: <code>\n</code> , <code>\r</code> , <code>\b</code> you must escape it with <code>\</code> too, except for the case when <code>-</code> is the first</p> <p>You can also include all characters matching Unicode properties (general <code>\p{EnumProperty=Value}</code> . (You can invert the test with <code>\P{Property}</code>.)</p> <p>For a list of valid Unicode property names, see Unicode Standard Annex <code>\p{Lu}</code> , <code>\p{Z}</code> , <code>\p{Symbol}</code> , <code>\p{Blk=Latin_1_Sup}</code> , and <code>\p{Blc</code></p>

As a shortcut for `\p{Block=Latin_1_Supplement}` , you can refer to `\p{InLatin_1_Supplement}` , `\p{InYijing_Hexagram_Symbols}` , and

A few extra properties are supported:

- `\p{Extended_Pictographic}` (see [UTS #35](#))
- `\p{EmojiPresentation=EmojiDefault}` (code points which have emoji presentation)
- `\p{EmojiPresentation=TextDefault}` (code points which have text emoji presentation)
- `\p{EmojiPresentation=Text}` (code points which have only text emoji presentation)

Property names are **case-insensitive**, and `_` and `-` are treated identically.

Here are a few examples:

```
WS : [ \n\u000D ] -> skip ; // same as [ \n\r]
```

```
UNICODE_WS : [\p{White_Space}] -> skip; // match all Unicode whitespace
```

```
ID : [a-zA-Z] [a-zA-Z0-9]* ; // match usual identifier specification
```

```
UNICODE_ID : [\p{Alpha}\p{General_Category=Other_Letter}]
```

```
EMOJI : [\u{1F4A9}\u{1F926}] ; // note Unicode code points
```

```
DASHBRACK : [ \- ]+ ; // match - or ] one or more times
```

```
DASH : [ - ] ; // match a single -, i.e., "any character"
```

'x'..'y'	Match any single character between range x and y, inclusively. E.g., 'a'..'z' matches any lowercase letter.
T	<p>Invoke lexer rule T; recursion is allowed in general, but not left recursion.</p> <pre>ID : LETTER (LETTER '0'..'9')* ;</pre> <p>fragment</p> <pre>LETTER : [a-zA-Z\u0080-\u00FF_] ;</pre>
.	<p>The dot is a single-character wildcard that matches any single character.</p> <pre>ESC : '\\ ' . ; // match any escaped \x character</pre>
{«action»}	<p>Lexer actions can appear anywhere as of 4.2, not just at the end of the rule. To execute a single action within the rule. To execute a single action afterwards:</p>

	<pre>END : ('endif' 'end') {System.out.println("found an end");}</pre> <p>The action conforms to the syntax of the target language. ANTLR cop like \$x.y as there is in parser actions.</p> <p>Only actions within the outermost token rule are executed. In other words, the lexer starts matching in STRING.</p>
{«p»}?	Evaluate semantic predicate «p». If «p» evaluates to false at runtime, language syntax. While semantic predicates can appear anywhere within the language, semantic predicates must precede lexer actions. See Predicates in the ANTLR Reference Grammar.
~x	<p>Match any single character not in the set described by x. Set x can be a character class, a character range, or a character set. ~ to match any character other than characters using ~[\r\n]*:</p> <pre>COMMENT : '#' ~[\r\n]* '\r'? '\n' -> skip ;</pre>

Just as with parser rules, lexer rules allow subrules in parentheses and EBNF operators: `?`, `*`, `+`. The `COMMENT` rule illustrates the `*` and `?` operators. A common use of `+` is `[0-9]+` to match integers. Lexer subrules can also use the nongreedy `?>` suffix on those EBNF operators.

Recursive Lexer Rules

ANTLR lexer rules can be recursive, unlike most lexical grammar tools. This comes in really handy when you want to match nested tokens like nested action blocks:

```
{...{...}...} .
```

```
lexer grammar Recur;
```

```
ACTION : '{' ( ACTION | ~[{}] )* '}' ;
```

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

Redundant String Literals

Be careful that you don't specify the same string literal on the right-hand side of multiple lexer rules. Such literals are ambiguous and could match multiple token types. ANTLR makes this literal unavailable to the parser. The same is true for rules across modes. For example, the following lexer grammar defines two tokens with the same character sequence:

```
lexer grammar L;
AND : '&' ;
mode STR;
MASK : '&' ;
```

A parser grammar cannot reference literal '&', but it can reference the name of the tokens:

```
parser grammar P;
options { tokenVocab=L; }
a : '&' // results in a tool error: no such token
    AND // no problem
    MASK // no problem
    ;
```

Here's a build and test sequence:

```
$ antlr4 L.g4 # yields L.tokens file needed by tokenVocab option in P.g4
$ antlr4 P.g4
error(126): P.g4:3:4: cannot create implicit token for string literal '&'
```

Lexer Rule Actions

An ANTLR lexer creates a `Token` object after matching a lexical rule. Each request for a token starts in `Lexer.nextToken`, which calls `emit` once it has identified a token. `emit` collects information from the current state of the lexer to build the token. It accesses fields `_type`, `_text`, `_channel`, `_tokenStartCharIndex`, `_tokenStartLine`, and `_tokenStartCharPositionInLine`. You can set the state of these with the various setter methods such as `setType`. For example, the following rule turns `enum` into an identifier if `enumIsKeyword` is false.

```
ENUM : 'enum' {if (!enumIsKeyword) setType(Identifier);} ;
```

ANTLR does no special `$x` attribute translations in lexer actions (unlike v3).

There can be at most a single action for a lexical rule, regardless of how many alternatives there are in that rule.

Lexer Commands

To avoid tying a grammar to a particular target language, ANTLR supports lexer commands. Unlike arbitrary embedded actions, these commands follow specific syntax and are limited to a few common commands. Lexer commands appear at the end of the outermost alternative of a lexer rule definition. Like arbitrary actions, there can only be one per token rule. A lexer command consists of the `->` operator followed by one or more command names that can optionally take parameters:

```
TokenName : «alternative» -> command-name
TokenName : «alternative» -> command-name («identifier or integer»)
```

An alternative can have more than one command separated by commas. Here are the valid command names:

- skip
- more
- popMode
- mode(x)
- pushMode(x)
- type(x)
- channel(x)

See the book source code for usage, some examples of which are shown here:

skip

A 'skip' command tells the lexer to get another token and throw out the current text.

```
ID : [a-zA-Z]+ ; // match identifiers
INT : [0-9]+ ; // match integers
NEWLINE: '\r'? '\n' ; // return newlines to parser (is end-statement
signal)
WS : [ \t]+ -> skip ; // toss out whitespace
```

mode(), pushMode(), popMode, and more

The mode commands alter the mode stack and hence the mode of the lexer. The 'more' command forces the lexer to get another token but without throwing out the current text. The token type will be that of the "final" rule matched (i.e., the one without a more or skip command).

```
// Default "mode": Everything OUTSIDE of a tag
COMMENT : '<!--' .*? '-->' ;
CDATA   : '<![CDATA[' .*? ']]>' ;
OPEN    : '<' -> pushMode(INSIDE) ;
...
XMLDeclOpen : '<?xml' S -> pushMode(INSIDE) ;
SPECIAL_OPEN: '<?' Name -> more, pushMode(PROC_INSTR) ;
// ----- Everything INSIDE of a tag -----
mode INSIDE;
CLOSE      : '>' -> popMode ;
SPECIAL_CLOSE: '?>' -> popMode ; // close <?xml...?>
SLASH_CLOSE : '/>' -> popMode ;
```

Also check out:

```
lexer grammar Strings;
LQUOTE : '"' -> more, mode(STR) ;
WS : [ \r\t\n]+ -> skip ;
mode STR;
STRING : '"' -> mode(DEFAULT_MODE) ; // token we want parser to see
TEXT : . -> more ; // collect more text for string
```

Popping the bottom layer of a mode stack will result in an exception. Switching modes with `mode` changes the current stack top. More than one `more` is the same as just one and the position does not matter.

type()

```
lexer grammar SetType;
tokens { STRING }
DOUBLE : '"' .*? '"' -> type(STRING) ;
SINGLE : '\'' .*? '\'' -> type(STRING) ;
WS : [ \r\t\n]+ -> skip ;
```

For multiple 'type()' commands, only the rightmost has an effect.

channel()

```
BLOCK_COMMENT
```

```

        : '/' '*' .*? '/' -> channel(HIDDEN)
        ;
LINE_COMMENT
    : '//' ~[\r\n]* -> channel(HIDDEN)
    ;

...
// -----
// Whitespace
//
// Characters and character constructs that are of no import
// to the parser and are used to make the grammar easier to read
// for humans.
//
WS : [ \t\r\n\f]+ -> channel(HIDDEN) ;

```

As of 4.5, you can also define channel names like enumerations with the following construct above the lexer rules:

```
channels { WSCHANNEL, MYHIDDEN }
```