## Contents

1 Library API  
2 Language definition guide  
3 Mode Reference  
4 CSS classes reference  
5 Style guide  
6 Plugins  
7 Recipes  
8 Language contributor checklist  
9 Building and Testing  
10 Maintainer’s guide  
11 Line numbers  
12 On requesting new languages  
13 Indices and tables
Contents:
Highlight.js exports a few functions as methods of the `hljs` object.

### 1.1 highlight(languageName, code, ignore_illegals, continuation)

Core highlighting function. Accepts a language name, or an alias, and a string with the code to highlight. The `ignore_illegals` parameter, when present and evaluates to a true value, forces highlighting to finish even in case of detecting illegal syntax for the language instead of throwing an exception. The `continuation` is an optional mode stack representing unfinished parsing. When present, the function will restart parsing from this state instead of initializing a new one. This is used internally for sublanguage support.

Note: `continuation` is NOT intended to support line-by-line highlighting because there is no requirement that a grammar handle linebreaks in any special way. It’s quite possible for a grammar to have a single mode/regex that matches MANY lines at once. This is not discouraged and entirely up to the grammar.

Returns an object with the following properties:

- **language**: language name, same as the name passed in `languageName`, returned for consistency with `highlightAuto`
- **relevance**: integer value representing the relevance score
- **value**: HTML string with highlighting markup
- **top**: top of the current mode stack
- **illegal**: boolean representing whether any illegal matches were found
- **code**: the original raw code
1.2 highlightAuto(code, languageSubset)

Highlighting with language detection. Accepts a string with the code to highlight and an optional array of language names and aliases restricting detection to only those languages. The subset can also be set with configure, but the local parameter overrides the option if set.

Returns an object with the following properties:

- language: detected language
- relevance: integer value representing the relevance score
- value: HTML string with highlighting markup
- second_best: object with the same structure for second-best heuristically detected language (may be absent)

1.3 fixMarkup(value) (deprecated as of 10.3)

fixMarkup is deprecated and will be removed entirely in v11.

Post-processing of the highlighted markup. Currently consists of replacing indentation TAB characters and using <br> tags instead of new-line characters. Options are set globally with configure.

Accepts a string with the highlighted markup.

1.4 highlightBlock(block)

Applies highlighting to a DOM node containing code.

This function is the one to use to apply highlighting dynamically after page load or within initialization code of third-party JavaScript frameworks.

The function uses language detection by default but you can specify the language in the class attribute of the DOM node. See the class reference for all available language names and aliases.

1.5 configure(options)

Configures global options:

- tabReplace: a string used to replace TAB characters in indentation.
- useBR: a flag to generate <br> tags instead of new-line characters in the output, useful when code is marked up using a non-<pre> container.
- classPrefix: a string prefix added before class names in the generated markup, used for backwards compatibility with stylesheets.
- languages: an array of language names and aliases restricting auto detection to only these languages.
- languageDetectRe: a regex to configure how CSS class names map to language (allows class names like say color-as-php vs the default of language-php, etc.)
- noHighlightRe: a regex to configure which CSS classes are to be skipped completely.

Accepts an object representing options with the values to updated. Other options don’t change
1.6 highlightAll()

Applies highlighting to all `<pre><code>...</code></pre>` blocks on a page. This can be called before or after the page's `onload` event has fired.

1.7 initHighlighting() (deprecated as of 10.6)

*Deprecated:* Please use `highlightAll()` instead.

Applies highlighting to all `<pre><code>...</code></pre>` blocks on a page.

1.8 initHighlightingOnLoad() (deprecated as of 10.6)

*Deprecated:* Please use `highlightAll()` instead.

Attaches highlighting to the page load event.

1.9 registerLanguage(languageName, languageDefinition)

Adds new language to the library under the specified name. Used mostly internally.

- `languageName`: a string with the name of the language being registered
- `languageDefinition`: a function that returns an object which represents the language definition. The function is passed the `hljs` object to be able to use common regular expressions defined within it.

1.10 unregisterLanguage(languageName)

Removes a language and its aliases from the library. Used mostly internally.

- `languageName`: a string with the name of the language being removed.

1.11 registerAliases(alias|aliases, {languageName})

Adds new language alias or aliases to the library for the specified language name defined under `languageName` key.

- `alias|aliases`: a string or array with the name of alias being registered
- `languageName`: the language name as specified by `registerLanguage`.
1.12 `listLanguages()`

Returns the languages names list.

1.13 `getLanguage(name)`

Looks up a language by name or alias.

Returns the language object if found, `undefined` otherwise.

1.14 `requireLanguage(name)` *(deprecated as of 10.4)*

This has been deprecated and will be removed in a future release. If you need this type of functionality use `getLanguage` with your own error handling. It is highly recommended that all inter-dependencies between grammars be handled at built-time, not run-time. This is what the core library now does.

Looks up a language by name or alias.

This should be used when one language definition depends on another. Using this function (vs `getLanguage`) will provide better error messaging when a required language is missing.

Returns the language object if found, raises a hard error otherwise.

1.15 `debugMode()`

Enables `debug/development` mode. This mode purposely makes Highlight.js more fragile! It should only be used for testing and local development (of languages or the library itself). By default “Safe Mode” is used, providing the most reliable experience for production usage.

For example, if a new version suddenly had a serious bug (or breaking change) that affected only a single language:

- **In Safe Mode**: All other languages would continue to highlight just fine. The broken language would appear as a code block, but without any highlighting (as if it were plaintext).

- **In Debug Mode**: All highlighting would stop when an error was encountered and a JavaScript error would be thrown.
2.1 Highlighting overview

Programming language code consists of parts with different rules of parsing: keywords like `for` or `if` don’t make sense inside strings, strings may contain backslash-escaped symbols like `\"`, and comments usually don’t contain anything interesting except the end of the comment.

In Highlight.js such parts are called “modes”.

Each mode consists of:

- starting condition
- ending condition
- list of contained sub-modes
- lexing rules and keywords
- …exotic stuff like another language inside a language

The parser’s work is to look for modes and their keywords. Upon finding them, it wraps them into the markup `<span class="...">...</span>` and puts the name of the mode (“string”, “comment”, “number”) or a keyword group name (“keyword”, “literal”, “built-in”) as the span’s class name.

2.2 General syntax

A language definition is a JavaScript object describing the default parsing mode for the language. This default mode contains sub-modes which in turn contain other sub-modes, effectively making the language definition a tree of modes.

Here’s an example:

```javascript
{
  case_insensitive: true, // language is case-insensitive
  keywords: 'for if while',
}
```
Usually the default mode accounts for the majority of the code and describes all language keywords. A notable exception here is XML in which a default mode is just a user text that doesn’t contain any keywords, and most interesting parsing happens inside tags.

### 2.3 Keywords

In the simple case language keywords can be defined with a string (space delimited) or array:

```json
{
  keywords: 'else for if while',
  // or with an array
  keywords: ['else', 'for', 'if', 'while']
}
```

Some languages have different kinds of “keywords” that might not be called as such by the language spec but are very close to them from the point of view of a syntax highlighter. These are all sorts of “literals”, “built-ins”, “symbols” and such. To define such keyword groups, the attribute `keywords` becomes an object, each property of which defines its own group of keywords:

```json
{
  keywords: {
    keyword: 'else for if while',
    literal: ['false', 'true', 'null']
  }
}
```

The group name becomes the class name in the generated markup, enabling different theming for different kinds of keywords.

To detect keywords, `highlight.js` breaks the processed chunk of code into separate words — a process called lexing. By default, “words” are matched with the regexp `/\w+`, and that works well for many languages. Different lexing rules can be defined by the magic `$pattern` attribute:
Note: The older `lexemes` setting has been deprecated in favor of using `keywords.$pattern`. They are functionally identical.

### 2.4 Sub-modes

Sub-modes are listed in the `contains` attribute:

```json
{
  keywords: '...',
  contains: [
    hljs.QUOTE_STRING_MODE,
    hljs.C_LINE_COMMENT,
    { ... custom mode definition ... }
  ]
}
```

A mode can reference itself in the `contains` array by using a special keyword `self`. This is commonly used to define nested modes:

```json
{
  className: 'object',
  begin: /\{/,
  end: /\}/,
  contains: [hljs.QUOTE_STRING_MODE, 'self']
}
```

Note: `self` may not be used in the root level `contains` of a language. The root level mode is special and may not be self-referential.

### 2.5 Comments

To define custom comments it is recommended to use a built-in helper function `hljs.COMMENT` instead of describing the mode directly, as it also defines a few default sub-modes that improve language detection and do other nice things.

Parameters for the function are:

```javascript
hljs.COMMENT(
  begin,   // begin regex
  end,     // end regex
  extra    // optional object with extra attributes to override defaults
            // (for example {relevance: 0})
)
```

### 2.6 Markup generation

Modes usually generate actual highlighting markup — `<span>` elements with specific class names that are defined by the `className` attribute:
Names are not required to be unique, it’s quite common to have several definitions with the same name. For example, many languages have various syntaxes for strings, comments, etc...

Sometimes modes are defined only to support specific parsing rules and aren’t needed in the final markup. A classic example is an escaping sequence inside strings allowing them to contain an ending quote.

For such modes, the className attribute should be omitted so they won’t generate excessive markup.

### 2.7 Mode attributes

Other useful attributes are defined in the *mode reference*.

### 2.8 Relevance

Highlight.js tries to automatically detect the language of a code fragment. The heuristics is essentially simple: it tries to highlight a fragment with all the language definitions and the one that yields most specific modes and keywords wins. The job of a language definition is to help this heuristics by hinting relative relevance (or irrelevance) of modes.

This is best illustrated by example. Python has special kinds of strings defined by prefix letters before the quotes: `r"...", u"..."`. If a code fragment contains such strings there is a good chance that it’s in Python. So these string modes are given high relevance:

On the other hand, conventional strings in plain single or double quotes aren’t specific to any language and it makes sense to bring their relevance to zero to lessen statistical noise:
The default value for relevance is always 1. When setting an explicit value typically either 10 or 0 is used. A 0 means this match should not be considered for language detection purposes. 0 should be used for very common matches that might be found in ANY language (basic numbers, strings, etc) or things that would otherwise create too many false positives. A 10 means “this is almost guaranteed to be XYZ code”. 10 should be used sparingly.

Keywords also influence relevance. Each of them usually has a relevance of 1, but there are some unique names that aren’t likely to be found outside of their languages, even in the form of variable names. For example just having `reinterpret_cast` somewhere in the code is a good indicator that we’re looking at C++. It’s worth to set relevance of such keywords a bit higher. This is done with a pipe:

```javascript
{ keywords: 'for if reinterpret_cast|10'
}
```

## 2.9 Illegal symbols

Another way to improve language detection is to define illegal symbols for a mode. For example, in Python the first line of a class definition `(class MyClass(object):)` cannot contain the symbol `{ or a newline. The presence of these symbols clearly shows that the language is not Python, and the parser can drop this attempt early.

Illegal symbols are defined using a single regular expression:

```javascript
{ className: 'class',
  illegal: '[${]'
}
```

## 2.10 Pre-defined modes and regular expressions

Many languages share common modes and regular expressions. These expressions are defined in `lib/modes.js` and should be used whenever possible.

## 2.11 Regular Expression Features

The goal of Highlight.js is to support whatever regex features JavaScript itself supports. You’re using real regular expressions, use them responsibly. That said, due to the design of the parser, there are some caveats. These are addressed below.

Things we support now that we did not always:

- look-ahead regex matching for `begin` (#2135)
- look-ahead regex matching for `end` (#2237)
- look-ahead regex matching for `illegal` (#2135)
- back-references within your regex matches (#1897)
- look-behind matching (when JS supports it) for `begin` (#2135)
Things we currently know are still issues:
  • look-behind matching (when JS supports it) for end matchers

2.12 Contributing

Follow the contributor checklist.
3.1 Types

Types of attributes values in this reference:

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>identifer</td>
<td>String suitable to be used as a JavaScript variable and CSS class name (i.e. mostly /^[A-Za-z0-9_]+/ )</td>
</tr>
<tr>
<td>regexp</td>
<td>String representing a JavaScript regexp. Note that since it’s not a literal regexp all backslashes should be repeated twice</td>
</tr>
<tr>
<td>boolean</td>
<td>JavaScript boolean: true or false</td>
</tr>
<tr>
<td>number</td>
<td>JavaScript number</td>
</tr>
<tr>
<td>object</td>
<td>JavaScript object: { ... }</td>
</tr>
<tr>
<td>array</td>
<td>JavaScript array: [ ... ]</td>
</tr>
</tbody>
</table>

3.2 Language Only Attributes

These attributes are only valid at the language level (ie, they may only exist on the top-most language object and have no meaning if specified in children modes).

3.2.1 name

- **type:** string

The canonical name of this language, ie “JavaScript”, etc.

3.2.2 case_insensitive

- **type:** boolean
Case insensitivity of language keywords and regexps. Used only on the top-level mode.

### 3.2.3 aliases

- **type**: array

A list of additional names (besides the canonical one given by the filename) that can be used to identify a language in HTML classes and in a call to `getLanguage`.

### 3.2.4 classNameAliases

- **type**: object

A mapping table of any custom class names your grammar uses and their supported equivalencies. Perhaps your language has a concept of “slots” that roughly correspond to variables in other languages. This allows you to write grammar code like:

```json
{
  classNameAliases: {
    slot: "variable",
    "message-name": "string"
  },
  contains: [ {
    className: "slot",
    begin: // ...
  } ]
}
```

The final HTML output will render slots with the CSS class as `hljs-variable`. This feature exists to make it easier for grammar maintainers to think in their own language when maintaining a grammar. For a list of all supported class names please see the [CSS class reference](#).

### 3.2.5 disableAutodetect

- **type**: boolean

Disables autodetection for this language.

### 3.2.6 compilerExtensions (USE WITH CAUTION)

- **type**: an array of compiler extensions ie: `''(mode, parentMode) -> {}''`

This allows grammars to extend the mode compiler to add their own syntactic sugar to make reading and writing grammars easier. **Note:** This is heavily dependent upon compiler internals and may NOT be stable from minor release to minor release. It is currently recommended only for 1st party grammars. The intention is that we use grammars to “test” out new compiler extensions and if they perform well promote them into the core library.

- **mode** The incoming mode object

- **parentMode** The parent mode of the mode (null for the top level language mode)

For example let’s look at a tiny well behaved extension to allow us to write `match` as sugar to better express the intent to “match a single thing, then end mode”.

14 Chapter 3. Mode Reference
Compiler extension functions return nothing. They are expected to mutate the mode itself.

## 3.3 Mode Attributes

### 3.3.1 className

- **type**: identifier

The name of the mode. It is used as a class name in HTML markup.

Multiple modes can have the same name. This is useful when a language has multiple variants of syntax for one thing like string in single or double quotes.

### 3.3.2 begin

- **type**: regexp

Regular expression starting a mode. For example a single quote for strings or two forward slashes for C-style comments. If absent, `begin` defaults to a regexp that matches anything, so the mode starts immediately.

### 3.3.3 match

- **type**: regexp

This is simply syntactic sugar for a `begin` when no `end` expression is necessary. It may not be used with `begin` or `end` keys (that would make no sense). It exists simply to help make grammars more readable.

```javascript
{
  className: "title",
  match: /Fish/
}
```

### 3.3.4 on:begin

- **type**: callback (matchData, response)

3.3. Mode Attributes 15
This callback is triggered the moment a begin match is detected. `matchData` includes the typical regex match data; the full match, match groups, etc. The `response` object is used to tell the parser how it should handle the match. It can be also used to temporarily store data.

- `response.data` - a simple object data store. Can be used for building more complex rules where the end rule is dependent on the content of begin, etc.
- `response.ignoreMatch()` - pretend as if this match never happened. The mode is not entered. Continues trying subsequent modes in the current mode’s `contains` list.

For an example of usage see `END_SAME_AS_BEGIN` in `modes.js`.

### 3.3.5 end

- **type**: `regexp`

Regular expression ending a mode. For example a single quote for strings or “$” (end of line) for one-line comments.

It’s often the case that a beginning regular expression defines the entire mode and doesn’t need any special ending. For example a number can be defined with `begin: "\b\d+"` which spans all the digits.

If absent, `end` defaults to a regexp that matches anything, so the mode ends immediately (after possibly matching any `contains` sub-modes).

Sometimes a mode can end not by itself but implicitly with its containing (parent) mode. This is achieved with `endsWithParent` attribute.

### 3.3.6 on:end

- **type**: `callback (matchData, response)`

This callback is triggered the moment an end match is detected. `matchData` includes the typical regex match data; the full match, match groups, etc. The `response` object is used to tell the parser how it should handle the match. It can also be used to retrieve data stored from a `begin` callback.

- `response.data` - a simple object data store. Can be used for building more complex rules where the end rule is dependent on the content of begin, etc.
- `response.ignoreMatch()` - pretend as if this match never happened. The mode is not entered. Continues trying subsequent modes in the current mode’s `contains` list.

For an example of usage see `END_SAME_AS_BEGIN` in `modes.js`.

### 3.3.7 beginKeywords

- **type**: `string`

Used instead of `begin` for modes starting with keywords to avoid needless repetition:

```json
{
    begin: '\b(class|interface)\b',
    keywords: 'class interface'
}
```

... can often be shortened to:
Unlike the `keywords` attribute, this one allows only a simple list of space separated keywords. If you do need additional features of `keywords` or you just need more keywords for this mode you may include `keywords` along with `beginKeywords`.

Note: `beginKeywords` also checks for a `. before or after the keywords and will fail to match if one is found. This is to avoid false positives for method calls or property accesses.

Ex. `class A { ... }` would match while `A.class == B.class` would not.

### 3.3.8 `endsWithParent`

- **type**: boolean

A flag showing that a mode ends when its parent ends.

This is best demonstrated by example. In CSS syntax a selector has a set of rules contained within symbols “{” and “}”. Individual rules separated by “;” but the last one in a set can omit the terminating semicolon:

```css
p {
  width: 100%; color: red
}
```

This is when `endsWithParent` comes into play:

```javascript
{
  className: 'rules', begin: /\{/, end: /\}/,
  contains: [
    {className: 'rule', /* ... */ end: ';', endsWithParent: true}
  ]
}
```

### 3.3.9 `endsWithParent`

- **type**: boolean

Forces closing of the parent mode right after the current mode is closed.

This is used for modes that don’t have an easily expressible ending lexeme but instead could be closed after the last interesting sub-mode is found.

Here’s an example with two ways of defining functions in Elixir, one using a keyword `do` and another using a comma:

```elixir
def foo :clear, list do
  :ok
end

def foo, do: IO.puts "hello world"
```

Note that in the first case the parameter list after the function title may also include a comma. And if we’re only interested in highlighting a title we can tell it to end the function definition after itself:
3.3.10 endSameAsBegin (deprecated as of 10.1)

**Deprecated:** *This attribute has been deprecated.* You should instead use the END_SAME_AS_BEGIN mode or use the on:begin and on:end attributes to build more complex paired matchers.

- **type:** boolean
  Acts as end matching exactly the same string that was found by the corresponding begin regexp.

For example, in PostgreSQL string constants can use “dollar quotes”, consisting of a dollar sign, an optional tag of zero or more characters, and another dollar sign. String constant must be ended with the same construct using the same tag. It is possible to nest dollar-quoted string constants by choosing different tags at each nesting level:

```
$foo$
...
$bar$ nested $bar$
...
$foo$
```

In this case you can’t simply specify the same regexp for begin and end (say, "\$[a-z]\$"), but you can use begin: "\$[a-z]\$" and endSameAsBegin: true.

3.3.11 lexemes (now keywords.$pattern)

- **type:** regexp
  A regular expression that extracts individual “words” from the code to compare against keywords. The default value is \w+ which works for many languages.

Note: It’s now recommended that you use keywords.$pattern instead of lexemes, as this makes it easier to keep your keyword pattern associated with your keywords themselves, particularly if your keyword configuration is a constant that you repeat multiple times within different modes of your grammar.

3.3.12 keywords

- **type:** object / string / array
  Keyword definition comes in three forms:
  - "for while if|0 else weird_voodoo|10 ... " – a string of space-separated keywords with an optional relevance over a pipe
highlight.js Documentation, Release 10.6.0

For detailed explanation see Language definition guide.

### 3.3.13 illegal

- **type**: regexp or array

A regular expression or array that defines symbols illegal for the mode. When the parser finds a match for illegal expression it immediately drops parsing the whole language altogether.

```javascript
{
  illegal: /%/,
  // or using an array
  illegal: [ /%/, /cookies/ ]
}
```

### 3.3.14 excludeBegin, excludeEnd

- **type**: boolean

Exclude beginning or ending lexemes out of mode’s generated markup. For example in CSS syntax a rule ends with a semicolon. However visually it’s better not to color it as the rule contents. Having `excludeEnd: true` forces a `<span>` element for the rule to close before the semicolon.

### 3.3.15 returnBegin

- **type**: boolean

Returns just found beginning lexeme back into parser. This is used when beginning of a sub-mode is a complex expression that should not only be found within a parent mode but also parsed according to the rules of a sub-mode.

Since the parser is effectively goes back it’s quite possible to create a infinite loop here so use with caution!

### 3.3.16 returnEnd

- **type**: boolean

Returns just found ending lexeme back into parser. This is used for example to parse JavaScript embedded into HTML. A JavaScript block ends with the HTML closing tag `</script>` that cannot be parsed with JavaScript rules. So it is returned back into its parent HTML mode that knows what to do with it.

Since the parser is effectively goes back it’s quite possible to create a infinite loop here so use with caution!

### 3.3.17 contains

- **type**: array

The list of sub-modes that can be found inside the mode. For detailed explanation see Language definition guide.
3.3.18 starts

- **type**: identifier

The name of the mode that will start right after the current mode ends. The new mode won’t be contained within the current one.

Currently this attribute is used to highlight JavaScript and CSS contained within HTML. Tags `<script>` and `<style>` start sub-modes that use another language definition to parse their contents (see `subLanguage`).

3.3.19 variants

- **type**: array

Modification to the main definitions of the mode, effectively expanding it into several similar modes each having all the attributes from the main definition augmented or overridden by the variants:

```json
{
    className: 'string',
    contains: ['self', hljs.BACKSLASH_ESCAPE],
    relevance: 0,
    variants: [
        {begin: //, end: //},
        {begin: /'/, end: /'/, relevance: 1}
    ]
}
```

Note: `variants` has very specific behavior with regards to `contains: ['self']`. Let’s consider the example above. While you might think this would allow you to embed any type of string (double or single quoted) within any other string, it does not allow for this.

The variants are compiled into two discrete modes:

```json
{
    className: 'string', begin: //, contains: ['self', ... ]
}
{
    className: 'string', begin: //', contains: ['self', ... ]
}
```

Each mode’s `self` refers only to the new expanded mode, not the original mode with variants (which no longer exists after compiling).

Further info: https://github.com/highlightjs/highlight.js/issues/826

3.3.20 subLanguage

- **type**: string or array

Highlights the entire contents of the mode with another language.

When using this attribute there’s no point to define internal parsing rules like `lexemes (now keywords.$pattern)` or `keywords`. Also it is recommended to skip `className` attribute since the sublanguage will wrap the text in its own `<span class="language-name">`.

The value of the attribute controls which language or languages will be used for highlighting:

- language name: explicit highlighting with the specified language
- empty array: auto detection with all the languages available
- array of language names: auto detection constrained to the specified set
3.3.21 skip

- **type**: boolean

Skips any markup processing for the mode ensuring that it remains a part of its parent buffer along with the starting and the ending lexemes. This works in conjunction with the parent's `subLanguage` when it requires complex parsing.

Consider parsing PHP inside HTML:

```html
<p><? echo 'PHP'; /* ?> */ ?></p>
```

The `?>` inside the comment should **not** end the PHP part, so we have to handle pairs of `/* .. */` to correctly find the ending `?>`:

```json
{
    begin: '/<\?/', end: '/\?>/',
    subLanguage: 'php',
    contains: [{begin: '/\*', end: '\*/', skip: true}]
}
```

Without `skip: true` every comment would cause the parser to drop out back into the HTML mode.
4.1 Stytable classes

The general purpose classes are intended to be used for any language, but the other classes may also be used when they are semantically correct. For example if you had a general purpose language that allowed inline URLs:

```javascript
var GOOGLE = 'https://www.google.com/
```

It would be wholly reasonable to use the `link` class for this, even though your language is not considered a “Markup” language.

<table>
<thead>
<tr>
<th>General purpose</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>keyword</td>
<td>keyword in a regular Algol-style language</td>
</tr>
<tr>
<td>built_in</td>
<td>built-in or library object (constant, class, function)</td>
</tr>
<tr>
<td>type</td>
<td>data type (in a language with syntactically significant types) (string, int, etc.)</td>
</tr>
<tr>
<td>literal</td>
<td>special identifier for a built-in value (true, false, null, etc.)</td>
</tr>
<tr>
<td>number</td>
<td>number, including units and modifiers, if any.</td>
</tr>
<tr>
<td>operator</td>
<td>operators: +, -, &gt;&gt;, !, ==</td>
</tr>
<tr>
<td>punctuation</td>
<td>aux. punctuation that should be subtly highlighted (parentheses, brackets, etc.)</td>
</tr>
<tr>
<td>regexp</td>
<td>literal regular expression</td>
</tr>
<tr>
<td>string</td>
<td>literal string, character</td>
</tr>
<tr>
<td>subst</td>
<td>parsed section inside a literal string</td>
</tr>
<tr>
<td>symbol</td>
<td>symbolic constant, interned string, goto label</td>
</tr>
<tr>
<td>class</td>
<td>class or class-level declaration (interfaces, traits, modules, etc.), typically includes a <code>title</code></td>
</tr>
<tr>
<td>function</td>
<td>function or method declaration</td>
</tr>
<tr>
<td>variable</td>
<td>variables</td>
</tr>
<tr>
<td>title</td>
<td>name of a class or a function at the place of declaration</td>
</tr>
<tr>
<td>params</td>
<td>block of function arguments (parameters) at the place of declaration</td>
</tr>
<tr>
<td>comment</td>
<td>comments</td>
</tr>
<tr>
<td>doctag</td>
<td>documentation markup within comments</td>
</tr>
<tr>
<td>Meta</td>
<td></td>
</tr>
</tbody>
</table>
### Table 1 – continued from previous page

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>meta</td>
<td>flags, modifiers, annotations, processing instructions, preprocessor directives, etc</td>
</tr>
<tr>
<td>meta-keyword</td>
<td>keyword or built-in within meta construct</td>
</tr>
<tr>
<td>meta-string</td>
<td>string within meta construct</td>
</tr>
</tbody>
</table>

#### Tags, attributes, configs

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>section</td>
<td>heading of a section in a config file, heading in text markup</td>
</tr>
<tr>
<td>tag</td>
<td>XML/HTML tag</td>
</tr>
<tr>
<td>name</td>
<td>name of an XML tag, the first word in an s-expression</td>
</tr>
<tr>
<td>builtin-name</td>
<td>s-expression name from the language standard library</td>
</tr>
<tr>
<td>attr</td>
<td>name of an attribute with no language defined semantics (keys in JSON, set names in .ini)</td>
</tr>
<tr>
<td>attribute</td>
<td>name of an attribute followed by a structured value part, like CSS properties</td>
</tr>
</tbody>
</table>

#### Text Markup

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>bullet</td>
<td>list item bullet</td>
</tr>
<tr>
<td>code</td>
<td>code block</td>
</tr>
<tr>
<td>emphasis</td>
<td>emphasis</td>
</tr>
<tr>
<td>strong</td>
<td>strong emphasis</td>
</tr>
<tr>
<td>formula</td>
<td>mathematical formula</td>
</tr>
<tr>
<td>link</td>
<td>hyperlink</td>
</tr>
<tr>
<td>quote</td>
<td>quotation or blockquote</td>
</tr>
</tbody>
</table>

#### CSS

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>selector-tag</td>
<td>tag selector</td>
</tr>
<tr>
<td>selector-id</td>
<td>#id selector</td>
</tr>
<tr>
<td>selector-class</td>
<td>.class selector</td>
</tr>
<tr>
<td>selector-attr</td>
<td>[attr] selector</td>
</tr>
<tr>
<td>selector-pseudo</td>
<td>:pseudo selector</td>
</tr>
</tbody>
</table>

#### Templates

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>template-tag</td>
<td>tag of a template language</td>
</tr>
<tr>
<td>template-variable</td>
<td>variable in a template language</td>
</tr>
</tbody>
</table>

#### diff

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addition</td>
<td>added or changed line</td>
</tr>
<tr>
<td>deletion</td>
<td>deleted line</td>
</tr>
</tbody>
</table>

### 4.1.1 A note on newer classes

Some classes have been added more recently and do not enjoy universal theme support. For themes without support, these items will simply not be highlighted. This doesn’t mean not to use them, only that they will be highlighted better as support improves over time.

A list of these classes:

- operator
- punctuation

### 4.1.2 Reserved classes

The below classes (ReasonML) are left here for documentation purposes but may not be used in other grammars because they are very poorly supported by all themes.

If you’d like to help out with the larger issue here:

- https://github.com/highlightjs/highlight.js/issues/2521
- https://github.com/highlightjs/highlight.js/issues/2500
### 4.2 Language names and aliases

The language names and aliases table has moved to SUPPORTED_LANGUAGES.md.

<table>
<thead>
<tr>
<th>ReasonML</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>pattern-match</td>
<td>pattern matching matchers</td>
</tr>
<tr>
<td>typing</td>
<td>type signatures on function parameters</td>
</tr>
<tr>
<td>constructor</td>
<td>type constructors</td>
</tr>
<tr>
<td>module-access</td>
<td>scope access into a module</td>
</tr>
<tr>
<td>module</td>
<td>module reference within scope access</td>
</tr>
</tbody>
</table>
5.1 Key principle

Highlight.js themes are language agnostic.

Instead of trying to make a rich set of highlightable classes look good in a handful of languages we have a limited set of classes that work for all languages.

Hence, there are two important implications:

- Highlight.js styles tend to be minimalistic.
- It’s not possible to exactly emulate themes from other highlighting engines.

5.2 Defining a theme

A theme is a single CSS defining styles for class names listed in the class reference. The general guideline is to style all available classes, however an author may deliberately choose to exclude some (for example, .attr is usually left unstyled).

You are not required to invent a separate styling for every group of class names, it’s perfectly okay to group them:

```
.hljs-string,
.hljs-section,
.hljs-selector-class,
.hljs-template-variable,
.hljs-deletion {
  color: #800;
}
```

Use as few or as many unique style combinations as you want.
5.3 Typography and layout dos and don’ts

Don’t use:
- non-standard borders/margin/paddings for the root container .hljs
- specific font faces
- font size, line height and anything that affects position and size of characters within the container

Okay to use:
- colors (obviously!)
- italic, bold, underlining, etc.
- image backgrounds

These may seem arbitrary at first but it’s what has shown to make sense in practice.

There’s also a common set of rules that has to be defined for the root container verbatim:

```css
.hljs {
    display: block;
    overflow-x: auto;
    padding: 0.5em;
}
```

5.4 .subst

One important caveat: don’t forget to style .subst. It’s used for parsed sections within strings and almost always should be reset to the default color:

```css
.hljs, .hljs-subst {
    color: black;
}
```

5.5 Contributing

You should include a comment at the top of the CSS file with attribution and other meta data if necessary. The format is free:

```*/
Fancy style (c) John Smith <email@domain.com>
*/
```

If you’re a new contributor add yourself to the authors list in AUTHORS.txt Also update CHANGES.md with your contribution.

Send your contribution as a pull request on GitHub.
Highlight.js supports plugins.

### 6.1 API

You can add a plugin via the `addPlugin` API.

```javascript
// a plugin can be a class
addPlugin(new SimplePlugin());
addPlugin(new MoreComplexPlugin(options));

// or simply a keyed object of functions
addPlugin(
    'after:highlightBlock': ({ block, result, text }) => {
        // ...
    },
    'after:highlightBlock': ({ block, result, text }) => {
        // ...
    });
```

#### 6.1.1 Class based plugins

This approach is useful for more complex plugins that need to deal with configuration options or managing state. Highlight.js will instantiate a single instance of your class and execute its callbacks as necessary.

```javascript
class DataLanguagePlugin {
    constructor(options) {
        self.prefix = options.dataPrefix;
    }

    'after:highlightBlock': ({ block, result, text }) {
        // ...
    }
}
```
6.1.2 Function based plugins

This approach is best for simpler plugins.

```javascript
hljs.addPlugin(new DataLanguagePlugin({ dataPrefix: 'hljs' }));
```

6.2 Callbacks

6.2.1 before:highlight({code, language})

This callback function is passed a context object with two keys:

- `code` The code to be highlighted.
- `language` The language grammar that should be used for highlighting.

Your plugin may modify either value and those new values will be used as input to the highlighting engine. If you add a `result` key to the object that result will be returned as the overall result and the internal highlighting code will never even be called.

If your plugin plans to make its own recursive calls to `highlight` you’ll need to manually handle this. Each time `highlight` is called your plugin callbacks will also be called - making it easy to get into an infinite loop. You’ll likely need to use a class based plugin and add a guard so that your plugin code is only triggered on the initial call to `highlight` and not on any internal calls your plugin itself is making.

Note: This callback does not fire from highlighting resulting from auto-language detection.

It returns nothing.

6.2.2 after:highlight(result)

This callback function is passed the `result` object after highlighting is complete. Your plugin may make any changes it desires to the result object and that will be the final return value of the initial call to `highlight`.

Note: This callback does not fire from highlighting resulting from auto-language detection.

It returns nothing.

6.2.3 after:highlightBlock({block, result, text})

Deprecated as of 10.7. Please use `after:highlightElement`. 
6.2.4 before:highlightBlock({block, language})

Deprecated as of 10.7. Please use before:highlightElement.

6.2.5 after:highlightElement({el, result, text})

This callback function is passed an object with two keys:

- **el**  The HTML element that’s been highlighted.
- **result**  The result object returned by highlight or highlightAuto.
- **text**  The raw text that was to be highlighted.

It returns nothing.

6.2.6 before:highlightElement({el, language})

This callback function is passed an object with two keys:

- **el**  The HTML element that will be highlighted.
- **language**  The language determined from the class attribute (or undefined).

It returns nothing.
Below is a collection of useful plugin “recipes” that you might find helpful.

### 7.1 data-language

Let’s say you’d like to track the language that was auto-detected via a data attribute. This might prove useful if you desired to add a dynamic label via CSS with :before, etc.

```javascript
hljs.addPlugin( {
    afterHighlightBlock: ({block, result}) => {
        // move the language from the result into the dataset
        block.dataset.language = result.language
    }
})
```
8.1 1. Know that you are creating a 3rd party grammar

Sadly, due to lack of maintainer time we no longer merge new languages grammars into the core library. Instead, the project works by encouraging 3rd party language grammar development by willing contributors ready to help and maintain them. We’re also happy to host those 3rd party language grammars at the highlightjs GitHub organization. Or you’re welcome to host yourself - we’re always happy to link to to new language grammars.

We also make it easy to build, maintain, and distribute 3rd party grammar modules so that Highlight.js can always be easily extended with new languages. Using a 3rd party language (for end users) is often as simple as just adding a single line of code to their JavaScript or build system.

We’d love to have your grammar as a 3rd party language module if you’d be willing to maintain it over time. It’s easy to develop and publish a 3rd party language module. If you already started work in the main source tree - it’s trivial to convert your existing work into a 3rd party module. (you pretty much just move your files into a new project folder)

8.2 2. Read extra/3RD_PARTY_QUICK_START.md

Next, read extra/3RD_PARTY_QUICK_START.md. This should provide you with a very high-level overview of the steps for creating a third-party language grammar for Highlight.js.

8.3 3. Create a language grammar definition file

The file defines a function accepting a reference to the library and returning a language object. The library parameter (hljs) is useful to access common modes and regexps. You should not immediately call this function (you only need to export it). Calling it is handled by the build process and details differ for different build targets.

```javascript
export default function(hljs) {
  return {
    name: "Language Name",
    (continues on next page)```
The name of the file is used as a short language identifier and should be usable as a class name in HTML and CSS. Typically you’ll place this file in your new grammar repository under `my_new_grammar/src/languages/`.

### 8.4 Add language metadata

At the top of the file there is a specially formatted comment with meta data processed by a build system. Meta data format is simply key-value pairs each occupying its own line:

```plaintext
/*
Language: Superlanguage
Requires: java.js, sql.js
Author: John Smith <email@domain.com>
Contributors: Mike Johnson <...@...>, Matt Wilson <...@...>
Description: Some cool language definition
Website: https://superlanguage.example.com
*/
```

- **Language** — the only required header giving a human-readable language name.
- **Requires** — a list of other language files required for this language to work. This make it possible to describe languages that extend definitions of other ones. Required files aren’t processed in any special way. The build system just makes sure that they will be in the final package in `LANGUAGES` object.

The meaning of the other headers should be pretty obvious.

### 8.5 Create a code example

The code example is used both to test language detection and for the demo page on https://highlightjs.org/. Put it in `test/detect/<language>/default.txt`.

Take inspiration from other languages in `test/detect/` and read testing instructions for more details.

### 8.6 Write a class reference if your class uses custom classes

A class reference document should typically be placed at the root of your language repository: `css-class-reference.md`

Describe shortly names of all meaningful modes used in your language definition.

Note: If you use custom classes please be aware that all the build-in themes are not going to support your custom classes and you should likely also distribute your own custom theme.

### 8.7 Request a repository at the highlight.js organization

*This is optional.* Of course you are free to host your repository anywhere you would like.
8.8 8. Create a pull request

Submit a PR to add your language to `SUPPORTED_LANGUAGES.md`. 
To use Highlight.js it is first necessary to build it for the environment where you plan to execute it: the browser, the Node.js server, etc.

### 9.1 TLDR

Often when contributing a pull-request it’s sufficient to build and test only the Node.js build. Our CI process will guarantee the browser build tests are all still green if you don’t run them during development.

```bash
npm run build
npm run test
```

The browser library must be built and tested separately:

```bash
npm run build-browser
npm run test-browser
```

### 9.2 Building

The build tool is written in JavaScript using node.js. Before running the script, make sure to have node installed and run `npm install` to get the dependencies.

The tool is located in `tools/build.js`. A few useful examples:

- Build for a browser using only common languages:

  ```bash
  node tools/build.js :common
  ```

- Build for node.js including all available languages:
node tools/build.js -t node

- Build two specific languages for debugging, skipping compression in this case:
  
  node tools/build.js -n python ruby

On some systems the node binary is named nodejs; simply replace node with nodejs in the examples above if that is the case.

The full option reference is available with the usual --help option.

The build result will be in the build/ directory.

### 9.3 Basic testing

The usual approach to debugging and testing a language is first doing it visually. You need to build highlight.js with only the language you're working on (without compression, to have readable code in browser error messages) and then use the Developer tool in tools/developer.html to see how it highlights a test snippet in that language.

A test snippet should be short and give the idea of the overall look of the language. It shouldn’t include every possible syntactic element and shouldn’t even make practical sense.

After you satisfied with the result you need to make sure that language detection still works with your language definition included in the whole suite.

Testing is done using Mocha and the files are found in the test/ directory. You can use the node build to run the tests from the command line with npm test after building. (Using npm run build_and_test you can build and then test with one command.)

**Note:** for Debian-based machine, like Ubuntu, you might need to create an alias or symbolic link for nodejs to node. The reason for this is the dependencies that are requires to test highlight.js has a reference to “node”.

Place the snippet you used inside the browser in test/detect/<language>/default.txt, build the package with all the languages for node and run the test suite. If your language breaks auto-detection, it should be fixed by improving relevance, which is a black art in and of itself. When in doubt, please refer to the discussion group!

### 9.4 Testing markup

You can also provide additional markup tests for the language to test isolated cases of various syntactic construct. If your language has 19 different string literals or complicated heuristics for telling division (/) apart from regexes (/ .. /) – this is the place.

A test case consists of two files:

- test/markup/<language>/<test_name>.txt: test code
- test/markup/<language>/<test_name>.expect.txt: reference rendering

To generate reference rendering use the Developer tool located at tools/developer.html. Make sure to explicitly select your language in the drop-down menu, as automatic detection is unlikely to work in this case.
9.5 Building and Testing with Docker

If you don’t want to install dependencies on your system, you can use the included Dockerfile to build a container that will add the source code and then deploy a web server for you to preview it. Specifically, after you finish with your changes you can build the container from the root of the repository:

```
docker build -t highlight-js .
```

And then run the container. You will need to expose port 80 on the host for the web interface, and note that we are running it in detached (-d) mode.

```
docker run -d --name highlight-js --rm -p 80:80 highlight-js
```

If your preference is for another port, you can do that too:

```
docker run -d --name highlight-js --rm -p 80:8080 highlight-js
```

Or you can skip binding a port if your intention is to interactively shell into the container to use it as a development environment.

```
docker run -d --name highlight-js --rm highlight-js
```

Whatever you choose, you can use `docker ps` to ensure that it’s running.

```
$ docker ps
CONTAINER ID IMAGE COMMAND CREATED STATUS PORTS NAMES
0e15a7c99adf highlight-js "docker-entrypoint.s..." 8 seconds ago Up 7 seconds 0.0.0.0:80->80/tcp highlight-js
```

Then, open up to `http://127.0.0.1/tools/developer.html` to see the developer page for preview. When you are done, clean up your container.

```
docker stop highlight-js
```

If you want a more advanced testing setup, you can bind the source folder when you run the container.

```
docker run -d --name highlight-js --volume $PWD/src:/var/www/html/src --rm -p 80:80
```

Then if you want to make changes, you can do so locally (the folder is bound as a volume), and execute a command to the container to trigger a rebuild:

```
docker exec highlight-js node tools/build.js :common
```

And then reload the page to see changes. When you finish, don’t forget to remove the container.
10.1 Commit policy

- Pull requests from outside contributors require a review from a maintainer.
- Maintainers should avoid working on a master branch directly and create branches for everything. A code review from another maintainer is recommended but not required, use your best judgment.

10.2 Release process

Releases (minor) typically happen on a 6-week schedule.

For major/minor releases you’ll be releasing from master. For patch releases you’ll be releasing from a stable branch, such as 10-stable. This allows ongoing development of new features to continue in isolation (in master) without those changes leaking into patch releases (which should focus only on fixing breaking changes).

The goal being that minor version series always get more stable over time and that patch releases do not add features.

- For patch releases: First switch to the associated stable branch (i.e., 10-stable)
- Update CHANGES.md with everything interesting since the last update.
- Update version numbers using the three-part x.y.z notation everywhere:
  - The header in CHANGES.md (this is where the site looks for the latest version number)
  - version attribute in package.json
  - version attribute in package-lock.json (run npm install)
  - release variable in docs/conf.py
  - All mentions of version number in the README.md
  - All mentions of version number in the README.ru.md
  - Update SECURITY.md version table
• Commit the version changes and tag the commit with the version number (10.4.0, no “v” prefix or anything like that)

• For major/minor releases: Create (or push push) the [major]-stable branch such as 10-stable

IE, the 10-stable branch should always point to the latest stable release in the 10 series.

• Push the commit the tag (git push && git push 10.4.0)

Pushing the tag triggers the update process which can be monitored at http://highlightjs.org/api/release/

When something didn’t work and it’s fixable in code (version numbers mismatch, last minute patches, etc), simply make another release incrementing the third (revision) part of the version number.

Miscellaneous:
Highlight.js’ notable lack of line numbers support is not an oversight but a feature. Following is the explanation of this policy from the current project maintainer (hey guys!):

One of the defining design principles for Highlight.js from the start was simplicity. Not the simplicity of code (in fact, it’s quite complex) but the simplicity of usage and of the actual look of highlighted snippets on HTML pages. Many highlighters, in my opinion, are overdoing it with such things as separate colors for every single type of lexemes, striped backgrounds, fancy buttons around code blocks and — yes — line numbers. The more fancy stuff resides around the code the more it distracts a reader from understanding it.

This is why it’s not a straightforward decision: this new feature will not just make Highlight.js better, it might actually make it worse simply by making it look more bloated in blog posts around the Internet. This is why I’m asking people to show that it’s worth it.

The only real use-case that ever was brought up in support of line numbers is referencing code from the descriptive text around it. On my own blog I was always solving this either with comments within the code itself or by breaking the larger snippets into smaller ones and describing each small part separately. I’m not saying that my solution is better. But I don’t see how line numbers are better either. And the only way to show that they are better is to set up some usability research on the subject. I doubt anyone would bother to do it.

Then there’s maintenance. So far the core code of Highlight.js is maintained by only one person — yours truly. Inclusion of any new code in Highlight.js means that from that moment I will have to fix bugs in it, improve it further, make it work together with the rest of the code, defend its design. And I don’t want to do all this for the feature that I consider “evil” and probably will never use myself.

This position is subject to discuss. Also it doesn’t stop anyone from forking the code and maintaining line-numbers implementation separately.
On requesting new languages

This is a general answer to requests for adding new languages that appear from time to time in the Highlight.js issue tracker and discussion group.

Highlight.js does not have a fundamental plan for implementing new languages - i.e., the core team doesn’t usually develop new languages. The core team instead focuses on parser development, bugs, and supporting the existing languages. They also currently does not have time to review, merge and maintain any additional languages (fixing bugs, dealing with issues, etc).

Instead, the project works by encouraging 3rd party language grammars from contributors willing to help develop and maintain them. We’re also happy to host those 3rd party language grammars at the highlight.js GitHub organization - no matter how obscure or weird. Or you’re welcome to host it yourself - we’re still happy to link to it.

This means that there’s no point in requesting a new language without also providing a 3rd party implementation (we’ll simply close “Please support language Xyz” issues with a link to this explanation). If you’d like to see a particular language available but cannot implement it, the best way to make it happen is to find another developer interested in doing so.

For more info on actually developing a language see our Language definition guide, and for information on how to properly package your 3rd party language module see Language contributor checklist.

Links:

- Code: https://github.com/highlightjs/highlight.js
- Discussion: http://groups.google.com/group/highlightjs
- Bug tracking: https://github.com/highlightjs/highlight.js/issues
CHAPTER 13

Indices and tables

- genindex
- modindex
- search