The codehighlight Package
Version 1.0

Alceu Frigeri*

November 2026

Abstract

This package enables the typesetting of verbatim code with syntax highlight. It allows for customization of all aspects related to syntax highlight, including the definition of what constitute a comment, a string, an identifier, etc. No attempt is made to grab some code in verbatim mode, this package relies on other packages for that, like *codeverbatim* [2] (its companion package) or the likes of *fancyvrb* [7] or *scontents* [3].

Contents

1 Introduction **2**

2 The codehighlight Package **2**
- 2.1 UTF8 and other active characters . . . 3
- 2.2 Display Settings . . . 3
- 2.3 Display commands . . . 4
- 2.4 Language and syntax highlight definitions . . . 5
- 2.5 Based on syntax and class remapping . . . 5
- 2.6 Internal structures and scan functions . . . 6
- 2.7 Package settings . . . 6
- 2.8 Base Format . . . 6
- 2.9 Highlight Sets . . . 7
- 2.10 Auxiliary Format Commands . . . 8
- 2.11 Keywords Sets . . . 8
- 2.12 Language Syntax . . . 9
- 2.13 Aliases . . . 12
- 2.14 Show Commands . . . 13
- 2.15 Pre-defined Sets . . . 14

3 Examples **17**
- 3.1 Preloading sets . . . 17
- 3.2 A minimal construct . . . 18
- 3.3 Customizing the language . . . 18
- 3.4 Further customization . . . 19
- 3.5 Using sequences . . . 19
- 3.6 Verbatim shortcuts . . . 20
  - 3.6.1 Custom settings . . . 20
  - 3.6.2 Custom highlight set . . . 20
  - 3.6.3 Custom shortcuts . . . 20
- 3.7 A floating environment . . . 21
- 3.8 Using listings with codeverbatim . . . 22

4 To do, Future plans **23**

*https://github.com/alceu-frigeri/codeverbatim

1 Introduction

The aim of this package (and its companion *codeverbatim* [2]) is to allow the creation of fully customizable verbatim environments. To achieve that, unlike other packages of the kind as *fancyvrb* [7], *scontents* [3] or *listings* [4], which have a fixed set of ways to collect/display a verbatim environment, *codeverbatim* and *codehighlight* split the action of grabbing some code in verbatim mode (package, *codeverbatim* [2]) from typesetting it with or without syntax highlight (this package *codehighlight*).

codehighlight offers ways to (syntax) highlight the code grabbed with *codeverbatim* [2]. It's somewhat inspired on the *listings* [4] functionality, allowing the definition of a custom syntax (for code highlight). Unlike *listings* [4], in *codehighlight* four aspects: (1) syntax definition, (2) keyword's sets, (3) highlight formatting and (4) base formatting, are kept separated, allowing a more flexible setup.

> ***Note:*** Although the two packages are though to be used together, they can be used separately. It's perfectly possible to use *codeverbatim* together with *listings* [4] package, or *codehighlight* [1] with other verbatim package, like *fancyvrb* [7] (using the macros `\SaveVerb` and `\UseVerb`) or *scontents* [3] (redefining the *verbatimsc* environment).

Lastly, this package is fully written in *expl3* [6], but given that one might want to use it in a LaTeX 2_ε context (either at package or document level), all public commands have a LaTeX 2_ε alias, and, for simplicity, they are listed together.

2 The codehighlight Package

After capturing some code in verbatim mode (for instance, using the package *codeverbatim* [2]) this provides commands to typeset them, highlighting some structures/keywords in the process.

> ***Note:*** This is not a pretty print or formatter package. That for, a full parser would be necessary, which is way beyond the scope of this package.

The specification of how and what will be highlighted is split in:

- Language syntax (⟨`lang`⟩ from 2.12), which defines how the code scanner can recognize elements as identifiers, comments, strings, etc. and associate those recognized elements with a formatting class.
- Keywords sets (⟨`kewy-set`⟩ from 2.11), which just defines sets of keywords (list), and associate them with a formatting class.
- Highlight sets (⟨`hlset-set`⟩ from 2.9), which maps a formatting class with a formatting code itself.
- Base format sets (⟨`basefmt-set`⟩ from 2.8) which sets the general formatting of the code to be displayed, like frame, background, etc.

All those aspects are decouple from each other, for instance one can use different sets of keywords with a given language, or have alternative highlight sets and use one or the other as desired.

As mentioned, the following display commands assumes the code was captured verbatim (for instance with *codeverbatim* [2]) and stored in an *expl3* [6] token list variable. The user is free to capture it in other ways, or process the code of said variable, in which case a few remarks are important:

- The end of line, line break mark is `^^J` (following the *expl3* [6] convention (see command `\tl_retokenize:n` in *"Reassigning category codes"* from *The LaTeX3 Interfaces* [6]).
- No character with the following category codes should appear (of course, the characters themselves can appear, just not with the following category codes):

catcode	name	usually	catcode	name	usually
0	escape	`\`	6	parameter	`#`
1	begin group	`}`	7	math superscript	`^`
2	end group	`{`	8	math subscript	`_`
3	math toggle	`$`	14	comment	`%`
4	alignment	`&`			

2.1 UTF8 and other active characters

Characters with character code greater then 127 are assumed to be active `utf8` characters and "aren't touched": the scanning routines will keep then as is, together, letting LaTeX 2_ε process them in the usual way.

On the other hand, characters with a character code below 128 are assumed to be non-active. In general, it's advisable to handle them before calling any of the display commands, either by turning them off before grabbing the code, or setting a custom environment with *codeverbatim* (see sections *"New Verbatim Environment"* and *"Characters Category Code and Common Keys"* from *codeverbatim* [2]), to turn them off or processing the grabbed code before displaying it.

The main problem that might arise from an active character (category code 13) with a character code below 128 is that the highlighting routines, besides the formatting code itself, will insert lots of breaking commands (e.g. `\penalty0\relax` commands) that, most likely, won't work well with whatever code was associated with the active character, most likely triggering low level kernel errors.

Of course, if the code associated with a given active character doesn't depends on *what is around*, it is safe to preserve them as active characters.

2.2 Display Settings

`\hlSettings` `\hlSettings {⟨keys⟩}`
`\codehl_settings:n` `\codehl_settings:n {⟨keys⟩}`

These will set (or change) the many display aspects. Note that, this is when the scan functions and variables associated with a language, keywords and highlight sets are created (see 2.6).

⟨keys⟩ can be any combination of:

`language` This will select the given language as the active one (See 2.12). If set to an empty value (*language = {}*) no language specific patterns will be recognized.

`hl set` This will select the given highlight set as the active one. See 2.9. If set to an empty value (*hl set = {}*) no syntax highlight will ensue.

`keywd set` This will select the given keyword set as the active one. See 2.11. If set to an empty value (*keywd set = {}*) no keyword will be recognized.

`base format` This will select the given base format set as the active one. See 2.8. If set to an empty value (*base format = {}*) then *std-basefmt* (see 2.8) will be selected as the base format.

> ***Note:*** All keys can be set to an empty value, but, if a non empty value is given, and the respective *language*, *hl set*, *keywords set* or *base format* doesn't exist, an error will be raised.
> ***Note:*** Initially, *language* and *keywd set* are set to empty (no definition), whilst, *base format* is set to *std-basefmt* and *hl set* is set to *std-hlset*.

`\hlSettingsDef` `\hlSettingsDef {⟨settings-id⟩}{⟨keys⟩}`
`\codehl_settings_def:nn` `\codehl_settings_def:nn {⟨settings-id⟩}{⟨keys⟩}`

This will create a settings set, ⟨settings-id⟩, and associate with it (for later use) a language ⟨lang⟩, highlight set ⟨hlset⟩, keyword set ⟨keyset⟩ and base format set ⟨basefmt⟩, to be used by `\hlSettingsSelect` or `\codehl_settings_select:n`. This is just a helper construct, in case one needs to, repeatedly, change over a few different setups.

⟨keys⟩ can be any combination of:

`based on language` {⟨lang-list⟩}

If set to empty, no language will be associated with ⟨settings-id⟩.

If a single value is given, ⟨lang⟩, it will test if ⟨lang⟩ exists, and associate it with ⟨settings-id⟩.

If ⟨lang-list⟩ is indeed a list, a language named ⟨settings-id_lang⟩ will be defined setting the *based on* key (see 2.5 and 2.12) to that list, and this language, ⟨settings-id_lang⟩, will be associated with ⟨settings-id⟩.

`based on hl set` {⟨hlset-list⟩}

If set to empty, no highlight set will be associated with ⟨settings-id⟩.

If a single value is given, ⟨hlset⟩, it will test if ⟨hlset⟩ exists, and associate it with ⟨settings-id⟩.

If ⟨hlset-list⟩ is indeed a list, a highlight set named ⟨settings-id_hlset⟩ will be defined setting the *based on* key (see 2.5 and 2.9) to that list, and this highlight set, ⟨settings-id_hlset⟩, will be associated with ⟨settings-id⟩.

`based on keywd set` {⟨keyset-list⟩}

If set to empty, no keyword set will be associated with ⟨settings-id⟩.

If a single value is given, ⟨keyset⟩, it will test if ⟨keyset⟩ exists, and associate it with ⟨settings-id⟩.

If ⟨keyset-list⟩ is indeed a list, a keyword set named ⟨settings-id_keyset⟩ will be defined setting the *based on* key (see 2.5 and 2.11) to that list, and this keyword set, ⟨settings-id_keyset⟩, will be associated with ⟨settings-id⟩.

`based on base format` {⟨basefmt-list⟩}

If set to empty, ⟨std-basefmt⟩ set will be associated with ⟨settings-id⟩.

If a single value is given, ⟨basefmt⟩, it will test if ⟨basefmt⟩ exists, and associate it with ⟨settings-id⟩.

If ⟨basefmt-list⟩ is indeed a list, a base format set named ⟨settings-id_basefmt⟩ will be defined setting the *based on* key (see 2.5 and 2.8) to that list, and this base format set, ⟨settings-id_basefmt⟩, will be associated with ⟨settings-id⟩.

`\hlSettingsSelect` `\hlSettingsSelect` {⟨settings-id⟩}
`\codehl_settings_select:n` `\codehl_settings_select:n` {⟨settings-id⟩}

This will call `\codehl_settings:n` with the ⟨language⟩, ⟨hl set⟩, ⟨keywd set⟩ and ⟨base format⟩ associated with ⟨settings-id⟩.

2.3 Display commands

`\hlDisplay` `\hlDisplay` [⟨keys⟩] {⟨code⟩}
`\hlDisplayV` `\hlDisplayV` [⟨keys⟩] {⟨tl-var⟩}
`\codehl_display:nn` `\codehl_display:nn` {⟨keys⟩} {⟨code⟩}
`\codehl_display:no` `\codehl_display:no` {⟨keys⟩} {⟨tl-var⟩}

These will display (typeset) the contents of ⟨code⟩ or ⟨tl-var⟩ (an *expl3* [6] token list variable) accordingly with the active language, highlight set, etc.

Note: The highlight routines expect ⟨code⟩ to be terminated by `^^J` , an end-of-line marker, as used by the command `\tl_retokenize:n` see *"Reassigning category code"* from *expl3* [6]. A quark loop will ensue otherwise.

Note: In it's default settings, a verbatim environment or short verbatim command created with *codeverbatim* [2], follow this convention.

⟨keys⟩ can be any combination of the keys from 2.8 (except the *based on* key) and the following:

`hl set` This will locally extend the active highlight set. The expected syntax is *hl set = {⟨hl-keys⟩}*, whereas ⟨hl-keys⟩ can be any combination of keys from 2.9 (except the *based on* key).

Note: This can only be used if a valid *hl set* was selected with `\hlSettings` or `\codehl_settings:n`.

`keywd set` This will locally extend the keywords set. The expected syntax is *keywd set = {⟨keywd-keys⟩}*, whereas ⟨keywd-keys⟩ can be any combination of keys from 2.11 (except the *based on* key).

Note: This can only be used if a valid *keywd set* was selected with `\hlSettings` or `\codehl_settings:n`.

2.4 Language and syntax highlight definitions

There are a few main cases to be taken care of:

- comments/strings → A language ⟨lang⟩ (see 2.12) defines the start/end patterns and associated ⟨class⟩, whilst a highlight set ⟨hlset-set⟩ (see 2.9) associates the related ⟨class⟩ with a format to be used.

  For example, when setting a language one can have: `comment = {3}{/*}{*/}` which will define a C++ style comment and associate `comment` ⟨class⟩ `3` with it. The format to be applied will come from a highlight set (see 2.9), like: `comment = {3}{\color {blue}}`.

  > ***Note:*** Comments can be discarded, or made *active*, meaning the comment text will be handled as normal LaTeX 2ε code.

- keywords → A keyword set, ⟨keyset-set⟩ (see 2.11), associates keywords with a ⟨class⟩, whilst a highlight set `hlset-set` (see 2.9) associates the related ⟨class⟩ with a format to be used.

  For example, when setting a keyword set, one can have: `keywd = {2}{some,words}` and `keywd = {3}{special}`. The format to be applied will come from a highlight set (see 2.9), like: `keywd = {2}{\bfseries }` and `keywd = {3}{\color {red}}`. That way the words *some* and *words* will appear in bold face, whilst *special* in red.

- identifiers → `language` (see 2.12) defines the delimiter ⟨start-pattern⟩ and associated ⟨id-name⟩, a ⟨keyset-set⟩ (see 2.11) associates keywords with a ⟨id-name⟩⟨class⟩, and ⟨hlset-set⟩ (see 2.9) associates the related ⟨id-name⟩⟨class⟩ with a format to be used.

  That's a special case of keywords, for example, when setting a language one can have: `identifier = {texcs}{\\}` which will define a LaTeX 2ε style identifier, associated with the ⟨id-name⟩ `texcs`.

  Then, when setting a keyword set, one can have: `texcs = {2}{cmdA,cmdB}` and `texcs = {3}{cmdC}`, and, finally, the format to be applied will come from a highlight set (see 2.9), like: `texcs = {2}{\bfseries }` and `texcs = {3}{\bfseries \color {blue}}`. That way the commands `\cmdA` and `\cmdB` will appear in bold face, whilst `\cmdC` in blue.

- tagged area keywords → `language` (see 2.12). When inside a `tagged area`, keywords associated with a `keywd` key, aren't recognized, but only those associated with `tag-name`. For instance `tagged area = { html } { < } { > } { non ~ tagged }` (`html` tag definition). Then, when setting a keyword set, one can have: `html = {2}{keywdA,keywdB}` and `html = {3}{keywdC}`, and, finally, the format to be applied will come from a highlight set (see 2.9), like: `html = {2}{\bfseries }` and `html = {3}{\bfseries \color {blue}}`. That way the keywords `keywdA` and `keywdB` will appear in bold face, whilst `keywdC` in blue, IF inside the tagged area.

  > ***Note:*** Note that, even though, all these aspects are related, they all (language sets, highlight sets, keyword sets and base format set) are decouple at definition time. Only when effectively displaying a given code that they will be "bring together", this allows (for instance) to have separated sets of keywords for a given language, and using one or another as needed.

2.5 Based on syntax and class remapping

In the following sections, a common key is `based on`, whose general syntax is:

`based on =` `{` ⟨base-set$_1$⟩ `,` ⟨base-set$_2$⟩ `...}`

And, ⟨base-set$_i$⟩ can be any combination of:

⟨base⟩	The ⟨base⟩ will be inherited, as is. No class re-mapping.
`R[int]`⟨base⟩	The ⟨base⟩ classes will be re-mapped, adding ⟨int⟩ to all sub-⟨base⟩ classes.
`S[int]`⟨base⟩	The ⟨base⟩ classes will be re-mapped, setting all sub-⟨base⟩ classes to ⟨int⟩.

> ***Note:*** A ⟨base⟩ can be based on other ⟨bases⟩. Keep in mind, though, no attempt is made to detect a circular definition, which will end exhausting LaTeX 2ε memory.

As briefly presented in the introduction above, a key concept of all sets (language, keywords, highlight) is the idea of a ⟨class⟩, an integer, which allows to have many formatting ⟨classes⟩ associated with each element.

Note: When processing the many sets (base format, see 2.5, highlight, see 2.9, keywords, see 2.11 or language, see 2.12) the *based on* keys are processed first in the definition order.
Moreover, the *based on* definitions can be nested. For instance, one can have a language *langA* based on languages *langB* and *langC*, and *langB* based on *langD*, and so on.

2.6 Internal structures and scan functions

Prior effectively displaying/highlighting some code, a set of scan functions (derived from the language definition, see 2.12) and auxiliary variables (derived from keywords sets, see 2.11, highlight sets, see 2.9 and base format, see 2.8) need to be set. Note though:

- Those scan functions and variables will always be locally defined. This allows to, for instance, have a local set of keywords extending a base set, but not changing it.

Depending on the specifics, setting up the needed scan functions (for a language) and auxiliary variables can be a time consuming operation. That's why setting those functions/variables is delayed as much as possible, eventually to the first call of `\hlSettings` (see 2.2). Usually this is ok, but, if `\hlSettings` is called from inside a local group (like inside an environment) this can lead to the undesirable side effect of rebuilding those functions/variables time and time again.

2.7 Package settings

`\hlPkgSettings` `\hlPkgSettings {⟨settings⟩}`
`\codehl_pkg_settings:n` `\codehl_pkg_settings:n {⟨settings⟩}`

This is meant to change some global settings. For now, ⟨settings⟩ can only be:

`info/note` (default: *note*) Possible values: *note*, *info* and *none* If set to *note* the *info/note* key from `\codehl_lang_def:nn` (see 2.12) and `\codehl_keyset_def:nn` (see 2.11) will be logged into the terminal and log file. If set to *info* it will be logged only into the log file. If set to *none*, they will be ignored. Note, the package's default is *info/note = note*.

2.8 Base Format

A base format defines general aspects of code highlighting, as framing, background color (if any), font, etc. By default, there is an initial set, *std-basefmt*, which only sets the *text width* to `\linewidth` and *wrap indent* to *2em*, besides *base format* to `\ttfamily\footnotesize` (the full setup can be seen with `\hlBasefmtShow{std-basefmt}`).

`\hlBasefmtDef` `\hlBasefmtDef {⟨basefmt-set⟩}{⟨keys⟩}`
`\codehl_basefmt_def:nn` `\codehl_basefmt_def:nn {⟨basefmt-set⟩}{⟨keys⟩}`

⟨keys⟩ can be any combination of:

`based on` *{ ⟨basefmt-set$_1$⟩ , ⟨basefmt-set$_2$⟩ ...}*
The definitions of ⟨basefmt-set$_i$⟩ set(s) are inherited. See 2.5 for details.

`info/note` {⟨note⟩}
⟨note⟩ will be logged into the log file. This is merely informational and can be suppressed with the command `\codehl_pkg_settings` (see 2.7).

Behaviour keys:

`gobble` (default: *0*) The number of characters to be gobble from each line prior any processing.

Note: There is no provision for utf8 chars here. It will fail badly if an utf8 character is present among the characters to be gobbled. The main purpose of this is to de-indent source code.

`inline` (default: *false*) Normally the code is displayed in vertical mode, constraining its width. With *inline* set to true, the code to be highlighted will be assumed to be short and displayed in its natural length.

> ***Note:*** A `^^J` found in the middle of ⟨code⟩ will result in a line break, except the one at the end of ⟨code⟩.

`add endmark` (default: *false*) This will add a `^^J` at the end of ⟨code⟩ (see 2.3) before it's further processed. This might be needed (to avoid a quark loop, see *"The l3quark module"* from *expl3* [6]) in case of verbatim code grabbed by other means than the *codeverbatim* [2] package, since the `\hlDisplay` command expects the verbatim code to be terminated with a `^^J` (following the convention used by `\tl_retokenize:n` see *"Reassigning category code"* from *expl3* [6]).

Format keys:

`base format` This will be applied in a local group in which all highlighted code will be displayed. This shouldn't absorb any token.

`show spaces` (default: *false*) If spaces are shown or not.

`text width` (default: *\linewidth*) This set's the text width of the displayed code.

`wrap indent` (default: *2em*) The indentation amount to be used in case of wrapping lines.

`background` Set's the background color to be used. Can be any color as defined by *xcolor*. By default no background is applied (equivalent to an empty argument).

Line numbers keys:

`numbers` (default: *none*) Possible values: *none*, *left* and *right*. Line numbers will be attached (or not) to each line.

`numbers format` Format to be used when displaying line numbers. The used command(s) can absorb a single parameter, namely the line number.

`numbers sep` (default: *2pt*) The separation between the displayed text and the line numbers.

Frame keys:

`rule width` (default: *0.25pt*) The width of the rules (if any) when framing the text.

`rule color` (default: *black*) The color of the rules (if any) when framing the text.

`left frame` (default: *false*) If a rule at the left should be present.

`right frame` (default: *false*) If a rule at the right should be present.

`top frame` (default: *false*) If a rule at the top should be present.

`bottom frame` (default: *false*) If a rule at the bottom should be present.

`frame sep` (default: *2pt*) The separation between the frames (if any) and the text.

`left sep` (default: *0pt*) Extra separation (added to *frame sep* between the left frame and the text.

`right sep` (default: *0pt*) Extra separation (added to *frame sep* between the right frame and the text.

`top sep` (default: *0pt*) Extra separation (added to *frame sep* between the top frame and the text.

`bottom sep` (default: *0pt*) Extra separation (added to *frame sep* between the bottom frame and the text.

2.9 Highlight Sets

A Highlight set defines the formatting to be applied to the many detected elements (as defined by language 2.12). The formats commands can absorb a single argument, namely the text/element detected and being formatted. By default, there is an initial set, *std-hlset*, which sets *comment* (classes 1 to 4) to *\itshape* , *string* (classes 1 to 4) to *\slshape* and *keywd* and *texcs* (classes 1 to 4) to *\slshape* (the full setup can be seen with `\hlHLsetShow{std-hlset}`). *std-hlset* is just for convenience, allowing for a quick and simple setup (like in 3.3).

\hlHLsetDef	\hlHLsetDef {⟨hlset-set⟩}{⟨keys⟩}
\codehl_hlset_def:nn	\codehl_hlset_def:nn {⟨hlset-set⟩}{⟨keys⟩}

All keys (except for *based on*) expect two arguments: {⟨class⟩}{⟨fmt-cmds⟩}.

⟨fmt-cmds⟩ can absorb a single argument. The ⟨class⟩ (an integer) allows to have distinct formats for the same structure (be it comments, strings, keywords, etc...).

⟨keys⟩ can be any combination of:

based on *{ ⟨hlset-ref$_1$⟩ , ⟨hlset-ref$_2$⟩ ...}*
The definitions of a base ⟨hlset-set$_i$⟩ set(s) are inherited. See 2.5 for details.

info/note {⟨note⟩}
⟨note⟩ will be logged into the log file. This is merely informational and can be suppressed with the command \codehl_pkg_settings (see 2.7).

comment {⟨class⟩}{⟨fmt-cmds⟩} To be applied to *comments*.

string {⟨class⟩}{⟨fmt-cmds⟩} To be applied to *strings*.

keywd {⟨class⟩}{⟨fmt-cmds⟩} To be applied to *keywords*.

emph {⟨class⟩}{⟨fmt-cmds⟩} To be applied to *keywords*, this is just an alternative way to format keywords (similar to *listings* [4] extra emphasis).

active {⟨class⟩}{⟨fmt-cmds⟩} To be applied to *active* comments.

⟨id-name⟩ {⟨class⟩}{⟨fmt-cmds⟩} To be applied to identifiers named ⟨id-name⟩. See 2.12 for details

⟨tag-name⟩ {⟨class⟩}{⟨fmt-cmds⟩} To be applied to (tagged) keywords inside a *tagged area*. See 2.12 for details

> ***Note:*** Since this is decoupled (at definition time) from language and keyword sets, any other *key* being used will be treated as a possible identifier (⟨id-name⟩ or ⟨tag-name⟩ see 2.12), and no error will be raised.
>
> ***Note:*** If you want to delete the format associated with any given *key* ⟨class⟩ pair, just redefine them as nothing, e.g., *keywd = {2}{}*.

2.10 Auxiliary Format Commands

\hlFmt	\hlFmt {⟨id⟩}{⟨class⟩}
\hlSetFmt	\hlSetFmt {⟨hlset-set⟩}{⟨id⟩}{⟨class⟩}
\codehl_hlset_fmt:nn	\codehl_hlset_fmt:nn {⟨id⟩}{⟨class⟩}
\codehl_hlset_fmt:nnn	\codehl_hlset_fmt:nnn {⟨hlset-set⟩}{⟨id⟩}{⟨class⟩}

These are meant to be used in the definition of ⟨fmt-cmds⟩, see 2.9. The versions with two parameters (⟨id⟩ and ⟨class⟩) use the current active set, whilst the ones with three parameters allow to reference the entries from a different highlight set. ⟨id⟩ can be any highlight set key: *comment*, *string*, *keywd*, *emph*, *active* or any *id-name* or *tag-name*. See 2.9.

> ***Note:*** Since this is decoupled (at definition time) from any language, highlight set and/or keyword set, no error will be raised if referencing a non-existent ⟨hlset-set⟩, ⟨id⟩ or ⟨class⟩.
>
> ***Note:*** These commands can be used anywhere, given that the corresponding highlight set was already processed (see 2.2 and 2.6), though an *active set* is only well defined in the context of display commands, see 2.3.

2.11 Keywords Sets

Sets of *detectable* keywords. These defines the list of keywords and associated formatting ⟨class⟩.

\hlKeysetDef	\hlKeysetDef {⟨keyset-set⟩}{⟨keys⟩}
\codehl_keyset_def:nn	\codehl_keyset_def:nn {⟨keyset-set⟩}{⟨keys⟩}

All keys (except for *based on*, see 2.5) expect two arguments: {⟨class⟩}{⟨keywd-list⟩}. Besides that, one can add an asterisk or an exclamation mark to each key, for instance: (a) *keywd** or

*emph**, in which case the list of keywords will be converted to lower case (using `\str_lowercase:n`), or (b) *keywd!* or *emph!*, in which case the list of keywords will be converted to upper case (using `\str_uppercase:n`).

The ⟨class⟩ (an integer) allows to have distinct sets of keywords, identifiers.

⟨keys⟩ can be any combination of:

`based on` { ⟨keyset-set$_1$⟩ , ⟨keyset-set$_2$⟩ ... }
The definitions of a base ⟨keyset-set$_i$⟩ set are inherited. See 2.5 for details.

`info/note` {⟨note⟩}
⟨note⟩ will be logged into the log file. This is merely informational and can be suppressed with the command `\codehl_pkg_settings` (see 2.7).

`keywd` {⟨class⟩}{⟨keywords list⟩} List of keywords.

`emph` {⟨class⟩}{⟨keywords list⟩} List of keywords to be formatted as *emph*.

⟨id-name⟩ {⟨class⟩}{⟨keywords list⟩} List of ⟨id-name⟩ (*identifiers*) keywords to be formatted as *id-name* in 2.9. See *identifiers* from 2.12 for details

⟨tag-name⟩ {⟨class⟩}{⟨keywords list⟩} List of keywords, ⟨tag-name⟩, to be formatted as ⟨*tag-name*⟩ (see 2.9) inside a *tagged area*. See 2.12 for details

> ***Note:*** Since this is decoupled (at definition time) from language and highlight format, any other *key* being used will be treated as a possible identifier (see 2.12), and no error will be raised.

`suppress` ⟨id-name⟩ {⟨class⟩}{⟨keywords list⟩} List of ⟨id-name⟩ keywords whose format (if already defined) will be suppressed. ⟨id-name⟩ can also be *keywd* or *emph*. ⟨class⟩ isn't used, it's needed just to preserve the same use pattern of the other keys.

> ***Note:*** Since this is decoupled (at definition time) from language and highlight format, no error will be raised if a keyword wasn't already defined.
>
> ***Note:*** Again, each keyword can be postfixed with (a) an asterisk, in which case the list of keywords will be converted to lower case (using `\str_lowercase:n`) or (b) an exclamation mark, in which case the list of keywords will be converted to upper case (using `\str_uppercase:n`). This is meant for case insensitive languages.

2.12 Language Syntax

The elements of a language (for instance, comments, strings) are defined in terms of their delimiters. There are 4 main cases:

- keywords: any combination of letters. That's *implicit* for all languages.
- Identifiers: keywords preceded by a marker (⟨start-pattern⟩ below), like TEX commands.
- Fully delimited elements (those with ⟨start-pattern⟩ and ⟨end-pattern⟩ below), like C++ comments starting with `/*` and ending with `*/`, or strings.
- Line delimited elements (those with ⟨start-pattern⟩ below), like C single line comments starting with `//`.

> ***Note:*** When defining the patterns, don't forget to escape TEX special characters, specially the ones with category code besides *other* (12) and *letter* (11): 0: escape itself (usually `\`), 1: begin-group (usually `{`), 2: end-group (usually `}`), 3: math shift (usually `$`), 4: alignment tab (usually `&`), 6: parameter (usually `#`), 7: superscript (usually `^`) , 8: subscript (usually `_`) , 14: comment (usually `%`).
> In fact, all characters can be escaped, for instance `{\a\d\:\$}` is, effectively, `{ad:$}`
>
> ***Attention:*** When processing the ⟨patterns⟩, no expansion will occur, and all characters in the given pattern will receive the category code other. For instance, given a pattern `{\cmdX}` the effective pattern will be `{cmdX}`.

All structures can be nested inside others, but only a few can *hold* them, namely: a *string* can be nested into *string* or *delimited string*, whilst a *comment* can be nested into *comment*, *comment line* or *comment line at*. The option *nestable* controls which ones can be detected (if nested).

In case of the *discard* variants, obviously a ⟨class⟩ isn't really needed (no formatting), but it's required so that its syntax is the same as the *comment* counter parts, which allows one to just add or remove a *discard* word, as needed.

When analysing the text to be highlighted, the listed patterns are grouped by their first letter, then the rest of the pattern is looked for in the *reverse* definition order, that means, for example, if one has two comment definitions, like `comment = {3}{/**}{**/}` followed by `comment = {1}{/*}{*/}` the `/**` will never be detected, always put the longest sequence last.

> ***Note:*** If two structures are defined with the same ⟨start-pattern⟩ the one defined last overwrites any previous definition for that ⟨start-pattern⟩.

`\hlLangDef` `\hlLangDef {⟨lang⟩}{⟨keys⟩}`
`\codehl_lang_def:nn` `\codehl_lang_def:nn {⟨lang⟩}{⟨keys⟩}`

In the following, ⟨options⟩ can any combination of the following:

`nestable` The construct will be detected if nested inside another likewise structure: comments inside comments, or strings/identifiers inside strings.

> ***Note:*** Any `comment` construct declared as `nestable` will be detected if inside `comment.comment line` or `comment line at`.
> A `string` or `identifier` construct declared as `nestable` will be detected if inside `string` or `delimited string`.

`tagged` The construct will only be detected inside a tagged area.

`non tagged` The construct will only be detected outside a tagged area.

> ***Note:*** If none is given, the construct will be detected by default (including inside a tagged area), but not inside other structures, like comments or strings.

⟨keys⟩ can be any combination of:

`based on` `{ ⟨lang-set$_1$⟩ , ⟨lang-set$_2$⟩ ...}`
The definitions of a base ⟨lang-set$_i$⟩ are inherited. See 2.5 for details.

`info/note` {⟨note⟩}
⟨note⟩ will be logged into the log file. This is merely informational and can be suppressed with the command `\codehl_pkg_settings` (see 2.7).

Keywords/identifiers behaviour keys:

`keywd in string` (default: `false`) If keywords (and/or identifiers) are to be detected (and formatted) inside strings.

> ***Note:*** In case of identifiers, they must be declared as `nestable` too.

`case sensitive` (default: `true`) Possible values: `true`, `false`, `lower case` or `upper case`. If keywords (and/or identifiers) are case sensitive or not, and how they have to be normalized (upper or lower case, if case insensitive) before comparing them with an existing keyword set.

> ***Note:*** If set to `false` or `lower case` (case insensitive) a 'keyword candidate' will be converted to lower case prior comparing it with any list of defined keywords, identifiers or tag names (see 2.11). If set to `upper case` (also case insensitive) a 'keyword candidate' will be converted to upper case. This implies that, it will only work IF the keywords lists are made of, respectively, lower or upper case keywords.

`alphanumeric` (default: `false`) If set to false (default) keywords and/or identifiers may only contain letters. If set to true they may start with `oletter` chars (besides letters) and may also contain `oletter` or `digits` chars.

> ***Note:*** This might interfere, in some cases, with other patterns, suppressing them if they start with one of the characters listed as `digits` or `oletter`. The best approach would be to just capture those characters as letter (see *"Characters Category Code and Common Keys"* and *"catcode commands"* from `codeverbatim` [2]).

`digits` {⟨dig-set⟩}
Sets the list of what a `digit` is. A regular expression composed of the range 0-9 plus ⟨dig-set⟩ will be constructed as `[0-9⟨dig-set⟩]` (see section *"Character classes"* from *The LaTeX3 Interfaces* [6]). This only applies if, and only if, `alphanumeric` is set to true.

`digits*` {⟨dig-set⟩}

Sets the list of what a *digit* is. A regular expression composed only of ⟨dig-set⟩ will be constructed as [⟨dig-set⟩] (see the section *"Character classes"* from *The LaTeX3 Interfaces* [6]). This only applies if, and only if, *alphanumeric* is set to true.

`oletter` {⟨o-set⟩}

Sets the list of what a *oletter* is. A regular expression composed only of ⟨o-set⟩ will be constructed as [⟨o-set⟩] (see the section *"Character classes"* from *The LaTeX3 Interfaces* [6]). This only applies if, and only if, *alphanumeric* is set to true.

> ***Note:*** Regarding *alphanumeric*, *digits* and *oletter*, it is slightly better (performance wise) to just set those as letter (category code 11) and avoid the extra overhead, setting *alphanumeric* to false.

Pattern keys:

`identifier` {⟨id-name⟩}{⟨start-pattern⟩}{⟨options⟩}

The keyword following ⟨start-pattern⟩ (including the ⟨start-pattern⟩ itself) will be formatted as defined by *id-name*⟨class⟩ (from ⟨hlset-set⟩).

`tagged area` {⟨tag-name⟩}{⟨start-pattern⟩}{⟨end-pattern⟩}{⟨options⟩}

In the region enclosed by ⟨start-pattern⟩ and ⟨end-pattern⟩ the usual keywords (the ones defined using the key *keywd* in a ⟨keyset-set⟩, see 2.11) won't be detected but only the ones declared as ⟨tag-name⟩. This aims at languages like *html* or *php*. If a *tagged area* is declared as *nestable* it can appear inside a string (for instance, like *php*).

`comment` {⟨class⟩}{⟨start-pattern⟩}{⟨end-pattern⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and ⟨end-pattern⟩, including them, will be formatted as defined by *comment*⟨class⟩ (from ⟨hlset-set⟩).

`comment line` {⟨class⟩}{⟨start-pattern⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and the end of line, will be formatted as defined by *comment*⟨class⟩ (from ⟨hlset-set⟩).

`comment line at` {⟨class⟩}{⟨start-pattern⟩}{⟨col⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and the end of line, will be formatted as defined by *comment*⟨class⟩ (from ⟨hlset-set⟩), but only if the first character appears at column ⟨col⟩.

`discard comment` {⟨class⟩}{⟨start-pattern⟩}{⟨end-pattern⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and ⟨end-pattern⟩, including them, will be discarded. The ⟨class⟩ is here just so this has the same construct as *comment*.

`discard comment line` {⟨class⟩}{⟨start-pattern⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and the end of line will be discarded. The ⟨class⟩ is here just so this has the same construct as *comment line*.

`discard comment line at` {⟨class⟩}{⟨start-pattern⟩}{⟨col⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and the end of line will be discarded, but only if the first character appears at column ⟨col⟩. The ⟨class⟩ is here just so this has the same construct as *comment line at*.

`active comment` {⟨class⟩}{⟨start-pattern⟩}{⟨end-pattern⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and ⟨end-pattern⟩, excluding them, will be: (1) executed as LaTeX 2$_\varepsilon$ commands. (2) formatted as defined by *active*⟨class⟩ (from ⟨hlset-set⟩).

`active comment line` {⟨class⟩}{⟨start-pattern⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and the end of line will: (1) executed as LaTeX 2$_\varepsilon$ commands. (2) formatted as defined by *active*⟨class⟩ (from ⟨hlset-set⟩).

`active comment line at` {⟨class⟩}{⟨start-pattern⟩}{⟨col⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and the end of line will: (1) executed as LaTeX 2$_\varepsilon$ commands. (2) formatted as defined by *active*⟨class⟩ (from ⟨hlset-set⟩). But only if the first character appears at column ⟨col⟩.

`string` {⟨class⟩}{⟨pattern⟩}{⟨escape-char⟩}{⟨options⟩}

Everything between ⟨pattern⟩ and ⟨pattern⟩, including them, will be formatted as defined by *string*⟨class⟩ (from ⟨hlset-set⟩). The ⟨escape-char⟩ assures the following character won't be interpreted as a pattern delimiter of any kind.

`delimited string` {⟨class⟩}{⟨start-pattern⟩}{⟨end-pattern⟩}{⟨escape-char⟩}{⟨options⟩}

Everything between ⟨start-pattern⟩ and ⟨end-pattern⟩, including them, will be formatted as defined by *string*⟨class⟩ (from ⟨hlset-set⟩). The ⟨escape-char⟩ assures the following character won't be interpreted as a pattern delimiter of any kind.

`replace` {⟨org-str⟩}{⟨replacement⟩}

Any occurrence of ⟨org-str⟩ will be replaced by ⟨replacement⟩. Note that ⟨org-str⟩ will be processed as any ⟨pattern⟩ in the previous keys, in particular, any space in it will be removed. Moreover, the scan routines will only look after those ⟨org-str⟩ after scanning for other patterns, the main implication: if a ⟨org-str⟩ starts with the same character used in any other pattern (comments, strings, etc.) this won't be recognized at all.

`replace*` {⟨org-str⟩}{⟨replacement⟩}

Any occurrence of ⟨org-str⟩ will be replaced by ⟨replacement⟩. Note that ⟨org-str⟩ will be processed as any ⟨pattern⟩ in the previous keys, in particular, any space in it will be removed. Unlike the previous key (without the asterisk) ⟨org-str⟩ will be treated as any other pattern, keep in mind that, in this case, a ⟨org-str⟩ might suppress the recognition of some other pattern (any previously defined one, starting with the same characters, but longer).

`suppress` {⟨pattern⟩}

The structure whose ⟨start-pattern⟩ equals ⟨pattern⟩ will be suppressed: no scan function will be generated for it.

> ***Note:*** It won't detect if the ⟨pattern⟩ corresponds, or not, to an already defined structure, given that the ⟨start-pattern⟩ might be inherited from a *based on* ⟨lang⟩.
>
> ***Note:*** Sequence matters: If, later on, a new structure is defined with the same ⟨start-pattern⟩, it will be normally processed (overwriting the *suppress* key).

2.13 Aliases

`\hlLangAlias` `\hlLangAlias` {⟨alias⟩}{⟨lang⟩}
`\codehl_lang_alias:nn` `\codehl_lang_alias:nn` {⟨alias⟩}{⟨lang⟩}

These commands define a set of aliases for a language. Notice, thought, that the internal representation of those aliases will always be exactly one level deep. For instance you can define 'B' as an alias for 'A' then 'C' as an alias to 'B', in which case, internally, 'C' will be just another alias for 'A'. More important, if, later on, 'B' is redefined as an alias for 'X', 'C' will remain an alias for 'A'.

> ***Note:*** An error will be raised if ⟨alias⟩ is an already defined as a language (not an alias).
>
> ***Note:*** The alias definition will be logged into the log file. This can be suppressed with the command `\codehl_pkg_settings` (see 2.7).

`\hlKeysetAlias` `\hlKeysetAlias` {⟨alias⟩}{⟨keyset⟩}
`\codehl_keyset_alias:nn` `\codehl_keyset_alias:nn` {⟨alias⟩}{⟨keyset⟩}

Same, for a keyword set.

`\hlHLsetAlias` `\hlHLsetAlias` {⟨alias⟩}{⟨hlset⟩}
`\codehl_hlset_alias:nn` `\codehl_hlset_alias:nn` {⟨alias⟩}{⟨hlset⟩}

Same, for a highlight set.

`\hlBasefmtAlias` `\hlBasefmtAlias` {⟨alias⟩}{⟨basefmt⟩}
`\codehl_basefmt_alias:nn` `\codehl_basefmt_alias:nn` {⟨alias⟩}{⟨basefmt⟩}

Same, for a base format set.

\hlSettingsAlias `\hlSettingsAlias {⟨alias⟩}{⟨settings-id⟩}`
\codehl_settings_alias:nn `\codehl_settings_alias:nn {⟨alias⟩}{⟨settings-id⟩}`

Same, for a settings set.

2.14 Show Commands

The following commands can be used to see what is currently defined and also which pre-defined sets (see 2.15) are registered for use.

\hlBasefmtShow `\hlBasefmtShow {⟨basefmt-set⟩}`
\codehl_basefmt_show:n `\codehl_basefmt_show:n {⟨basefmt-set⟩}`

This will write to the terminal (raising an error message) the effective ⟨basefmt-set⟩, after applying defaults and recovering all *based on* definitions.

\hlHLsetShow `\hlHLsetShow {⟨hlset-set⟩}`
\codehl_hlset_show:n `\codehl_hlset_show:n {⟨hlset-set⟩}`

This will write to the terminal (raising an error message) the effective ⟨hlset-set⟩ (after recovering all *based on* definitions).

\hlKeysetShow `\hlKeysetShow {⟨keyset-set⟩}`
\codehl_keyset_show:n `\codehl_keyset_show:n {⟨keyset-set⟩}`

This will write to the terminal (raising an error message) the effective ⟨keyset-set⟩ (after recovering all *based on* definitions).

\hlLangShow `\hlLangShow {⟨lang⟩}`
\codehl_lang_show:n `\codehl_lang_show:n {⟨lang⟩}`

This will write to the terminal (raising an error message) the effective ⟨lang⟩ definitions (after recovering all *based on* definitions).

\hlSettingsShow `\hlSettingsShow {⟨settings-id⟩}`
\codehl_settings_show:n `\codehl_settings_show:n {⟨settings-id⟩}`

This will write to the terminal (raising an error message) the ⟨settings-id⟩ definition. If called with an empty argument, `\hlSettingsShow{}`, it will log the currently active settings.

\hlShow `\hlShow {⟨keys⟩}`
\codehl_show:n `\codehl_show:n {⟨keys⟩}`

where ⟨keys⟩ can be any combination of the following:

`registered lang` This will log, all pre-defined language sets.

`registered keyset` This will log, all pre-defined keyword sets.

`registered hlet` This will log, all pre-defined highlight sets.

`registered basefmt` This will log, all pre-defined base format sets.

`registered settings` This will log, all pre-defined settings sets.

Note: In case of the *registered* keys, each entry will includes the file name where the sets can be found.

`defined lang` This will log, the name of all currently defined language sets.

`defined keyset` This will log, the name of all currently defined keyword sets

`defined hlset` This will log, the name of all currently defined highlight sets.

`defined basefmt` This will log, the name of all currently defined base format sets.

`defined settings` This will log, the name of all currently defined settings sets.

Note: When listing these *defined* sets, in case of an alias, they will point to the effective set name.

`lang show` {⟨lang-list⟩}
: This expects a comma separated list of languages, ⟨lang-list⟩, and it will call `\codehl_lang_show:n` for each one. Effectively logging each language structure.

`lang show all`
: This will call `\codehl_lang_show:n` for every currently defined language. Logging them all.

`keyset show` {⟨keyset-list⟩}
: This expects a comma separated list of keyword sets, ⟨keyset-list⟩, and it will call `\codehl_keyset_show:n` for each one. Effectively logging each keyword set.

`keyset show all`
: This will call `\codehl_keyset_show:n` for every currently defined keyword set. Logging them all.

`hlset show` {⟨hlset-list⟩}
: This expects a comma separated list of highlight sets, ⟨hlset-list⟩, and it will call `\codehl_hlset_show:n` for each one. Effectively logging each highlight set.

`hlset show all`
: This will call `\codehl_hlset_show:n` for every currently defined highlight set. Logging them all.

`basefmt show` {⟨basefmt-list⟩}
: This expects a comma separated list of base format sets, ⟨basefmt-list⟩, and it will call `\codehl_basefmt_show:n` for each one. Effectively logging each base format set.

`basefmt show all`
: This will call `\codehl_basefmt_show:n` for every currently defined base format set. Logging them all.

`settings show` {⟨settings-id-list⟩}
: This expects a comma separated list of settings sets, ⟨settings-id-list⟩, and it will call `\codehl_settings_show:n` for each one. Effectively logging each settings set. Additionally, it will log the currently active settings.

`settings show all`
: This will call `\codehl_settings_show:n` for every currently defined settings set. Logging them all. Additionally, it will log the currently active settings.

2.15 Pre-defined Sets

There is a set of pre-defined languages known by *codehighlight*, but since it would be wasteful to load all definitions all the time, those language definitions are *stored* in auxiliary files.

Those auxiliary files will be loaded as needed, for instance when one tries to expand the definitions of a pre-defined language (`\codehl_lang_def:nn`, see 2.12), or defines a language based on a pre-defined one (key *based on*, see 2.5), or sets the display defaults (`\codehl_settings:n`, see 2.2).

> ***Note:*** File loading, can only occurs in the preamble. An error will be raised otherwise.
>
> ***Attention:*** When defining a language, keyword set, highlight set, all definitions are local. This allows to, for instance, locally expand a keyword list, but it also has a side effect that, if an auxiliary file is loaded in a local group, the definitions will be lost.

The following commands, force the load of the needed auxiliary files, and should be used in the preamble. Note, though, that there isn't a one-to-one relationship between auxiliary packages and language/keyword sets, since some auxiliary packages might entice more than one language definition. Lastly, those commands keep track of which auxiliary files were already loaded, so it's safe to call them more than once for the same language/keyword set.

`\hlLangLoad` `\hlLangLoad` {⟨lang-list⟩}
`\codehl_lang_preload:n` `\codehl_lang_preload:n` {⟨lang-list⟩}

This will include the needed auxiliary packages. Whereas ⟨lang-list⟩ is a comma separated list of languages.

`\hlKeysetLoad` `\hlKeysetLoad` {⟨keyset-list⟩}
`\codehl_keyset_preload:n` `\codehl_keyset_preload:n` {⟨keyset-list⟩}

Similarly, whereas ⟨keyset-list⟩ is a comma separated list of keyset sets.

`\hlHLsetLoad` `\hlHLsetLoad {⟨hlset-list⟩}`
`\codehl_hlset_preload:n` `\codehl_hlset_preload:n {⟨hlset-list⟩}`

Similarly, whereas ⟨hlset-list⟩ is a comma separated list of hlset sets.

`\hlBasefmtLoad` `\hlBasefmtLoad {⟨basefmt-list⟩}`
`\codehl_basefmt_preload:n` `\codehl_basefmt_preload:n {⟨basefmt-list⟩}`

Similarly, whereas ⟨basefmt-list⟩ is a comma separated list of basefmt sets.

`\hlSettingsLoad` `\hlSettingsLoad {⟨settings-list⟩}`
`\codehl_settings_preload:n` `\codehl_settings_preload:n {⟨settings-list⟩}`

Similarly, whereas ⟨settings-list⟩ is a comma separated list of settings sets.

> ***Note:*** There is, currently, no predefined ⟨hlset⟩, ⟨basefmt⟩ or ⟨settings⟩ sets, besides ⟨std-basefmt⟩ and ⟨std-hlset⟩ (which are always defined). This is just for command's symmetry, or in case someone asks for a feature similar to *listings* [4] *.prf* files (user predefined preferences).

The following table contain the current list of known languages and keyword sets. A few remarks, first:

- Languages like Fortran also have a pre-defined alias (in this case, just Fortran) normally pointing to the 'most recent' definition. Note: if a new variant is introduced, the alias shall point to it.
- In most cases, when a language has multiple predefined sets, the last one (normally associated with the last version of them) is the more complete set, build upon the previous ones.
- In all cases, `\hlLangShow` and/or `\hlKeysetShow` can be used to verify the effective definitions.
- Some keyword's sets, ⟨keyset⟩, got split, either by an identifier (like *texcs* for TEX or *php* for php) or classes. To document all fine details would be prone to error. In doubt, double check with `\hlKeysetShow`.
- Languages belonging to the same family are, in general, hold together (for instance, the many FORTRAN keyword sets definitions).
- In most cases, the keyword sets are complete (to the author's knowledge), but in some cases it was needed to make some choices: the most glaring example *elisp.fullset* this is a separated set (single file) with nearly 6000 keywords, which where extracted from the documentation of it. Obviously, no practical way to verify them all, and (at best) questionable. In some other cases, it wasn't practical to have comprehensive lists, so the 'most common/usual' (fuzzy definition, prone to errors) were included.

Language	Language sets	Keywords sets	Alias / Obs.
C	C.89 C.99 C.11 C.17 C.23	C.89 C.99 C.11 C.17 C.23	C → C.23
C++	C++.98 C++.11 C++.20 C++.23	C++.98 C++.11 C++.20 C++.23	C++ → C++.23
Csharp	Csharp.1 Csharp.6 Csharp.11 Csharp.13	Csharp.1 Csharp.6 Csharp.11 Csharp.13	Csharp → Csharp.13
java	java.SE25	java.SE25	java → java.SE25
python	python.2.7 python.3.0 python.3.6 python.3.12	python.2.7 python.3.0 python.3.6 python.3.12	python → python.3.12
SQL	SQL.92 SQL.23 PL/SQL	SQL.92 SQL.23 PL/SQL SQL.PostgreSQL SQL.MySQL	SQL → SQL.23
perl	perl	perl	
lua	lua.5.5	lua.5.5	lua → lua.5.5
html	html.5	html.5	html → html.5
php	php.8	php.8 php.mysqli php.pgsql php.xml-json php.date	php → php.8
JavaScript	JavaScript	JavaScript	
scala	scala.3.10	scala.3.10	scala → scala.3.10
VisualBasic	VB.6 VB.8 VB.17	VB.6 VB.8 VB.12 VB.17	VisualBasic → VB.17 VisualBasicClassic → VB.6
Delphi	Delphi.7 Delphi.13	Delphi.7 Delphi.XE8 Delphi.13	Delphi → Delphi.13

Language	Language sets	Keywords sets	Alias / Obs.
Modula-2	Modula-2	Modula-2	
Oberon	Oberon Oberon+	Oberon Oberon+	
COBOL	COBOL.74 COBOL.85 COBOL.02	COBOL.74 COBOL.85 COBOL.02	COBOL → COBOL.02
FORTRAN	FORTRAN.77 FORTRAN.95 FORTRAN.03 FORTRAN.18	FORTRAN.77 FORTRAN.95 FORTRAN.03 FORTRAN.18	FORTRAN → FORTRAN.18
Rust	Rust.1.99	Rust.1.99 Rust.1.99.core Rust.1.99.alloc Rust.1.99.std Rust.1.99.all	Rust → Rust.1.99
R	R	R	
Ada	Ada.95 Ada.22	Ada.95 Ada.22	Ada → Ada.22
julia	julia.1.x	julia.1.x	julia → julia.1.x
Ruby	Ruby.4	Ruby.4	Ruby → Ruby.4
lisp	CL elisp	CL elisp elisp.fullset	CommonLisp → CL
awk	awk.posix awk.gnu	awk.posix awk.gnu	
postscript	postscript.level3	postscript.level3	postscript → postscript.level3
MetaFont	MetaFont	MetaFont	
MetaPost	MetaPost	MetaPost	
TeX	TeX	TeX.primites TeX.pdfTeX TeX.luaTeX TeX.common	
TeX	TeX	TeX.ams TeX.babel TeX.bibtex TeX.hyperref TeX.kernel TeX.pgf TeX.pgfplots TeX.xpacks	from *codedescribe*
TeX	TeX.l3	TeX.l3ams TeX.l3bibtex TeX.l3exp TeX.l3kernel TeX.l3pgf	from *codedescribe*. cmds contain `_`
TeX	TeX.l3sign	TeX.l3amssign TeX.l3bibtexsign TeX.l3expsign TeX.l3kernelsign TeX.l3pgfsign	from *codedescribe*. cmds contain `_:`

A few closing notes/remarks:

- In case of errors, the reader is invited to open an issue at `https://github.com/alceu-frigeri/codeverbatim`.
- "In general", when a language has single line comments mixed with multi line ones, the single line are defined as class 1, and the multi line as class 2. This allows (if necessary) to differentiate one from another. Same for strings, usually, the most common cases (a string delimited by `"` or `'`) are defined as class 1, whilst others as class 2.
- All subject to change: at the end, it should be up to those who use those languages more regularly, what sets (and classes) are more reasonable.
- a few languages have "tagged areas". When inside a tagged area, normal keywords aren't recognized but only the "tagged ones" (see 2.4). In particular *html* has a tagged area defined as *tagged area = { html } { < } { > } { non tagged }*. The *non tagged* assures it won't be recognized inside another tagged area. Moreover, the associated *keyset* defines keys associated with the identifier *html* (for instance, an usual html property like *div*, won't be recognized as a html keyword outside a tagged area).
- *php* is another such language (from it's definition: *tagged area = { php } { <?php } { ?> } { nestable }*) with a twist: it is "based on" the definition of *html*, meaning any valid *html* construct will also be recognized, besides *php* ones, furthermore they can appear inside *html* tagged areas, including *html* strings (the *nestable* in the *php* definition). Finally, note that, *html* constructs won't be recognized if they appear inside a *php* tagged area (the *non tagged* in the *html* definition).
- *SQL* and *COBOL* both are case insensitive, normalizing the keywords in upper case. This might become relevant in case they get used as the base for an extended ⟨keyset⟩ and used in a case sensitive context. All other case insensitive languages (currently *FORTRAN*, *Ada*, *html*, *VB* and *Delphi*) are normalized using lower case keywords. Why such exception? conventions. Even though they are case insensitive, it's usual to write their commands in upper case. That way, if they get used in a case sensitive context, the convention will hold.

- *PL/SQL* is defined as alphanumeric, allowing keywords with a `%`, more precisely, from it's definition: `oletter = { \% }` . Note though, that, when expanding the *PL/SQL* ⟨keyset⟩ it's mandatory to change the category code of `%` to other. `\hlKeysetDef` absorbs the keywords list as "other", therefore `\%` is different from `%`. Better said, if one tries `\hlKeysetDef { PL/SQL } { keywd = {2} {SQL\%word}}` instead of `\hlKeysetDef { PL/SQL } { keywd = {2} { SQL%word } }` one will end with a keyword `SQL\%word` instead of `SQL%word`. And for the last one, it's necessary to change the category code of a `%` first.

- TEX, there are three language definitions: *TeX*, *TeX.l3* and *TeX.l3sign*. They are basically the same (usual LATEX conventions). All three have `%` denoting a comment line, and an 'identifier' named *texcs* (more precisely *identifier = { texcs } { \\ } { }*). *TeX.l3* declares keywords as being alphanumeric, allowing for an underscore. *TeX.l3sign* also declares keywords as alphanumeric, allowing for underscore and a colon to be part of a keyword. These last two are meant to document *expl3* commands, though it is slightly more efficient to collect a sample code (see *"Characters Category Code and Common Keys"* from *codeverbatim* [2]) changing the catcode of these characters (namely underscore and colon) to letter and just use the language definition *TeX*.

- *TeX.primitives* contains all LATEX 2_ε primitives. *TeX.pdfTeX* is based on *TeX.primitives* adding pdflatex specific ones. *TeX.luaTeX* is based on *TeX.primitives* adding lualatex specific ones. *TeX.common* contains the most common user level, LATEX 2_ε, commands (but no *texcs* keywords from the previous sets). Moreover, *TeX.pdfTeX* and *TeX.luaTeX* are the only set the defines *texcs* keywords of class 2, all others define sets of class 1 (this allows to easy re-mapping when using the *based on* key (see following example).

- The other *TeX.* sets were programmatically extracted from the sources. The many *TeX.l3** and *TeX.l3*sign* contain *expl3* valid names, take note that, for instance, *TeX.l3ams* and *TeX.l3amssign* are fundamentally the same, the former containing the 'base name' (without the colon and command signature), whilst the last one contains all found commands with respective signatures (those files are large).

  They are all separated sets. If one wants to use one (or more) of them together with the previous ones, it is necessary to declare a new ⟨keyset⟩ based on the many ones the reader might want (see 2.5 for details). For instance:

```
\hlKeysetDef{TeX.mydef}
  {
    based on =
      {
        TeX.primitives  ,
        S[2]TeX.common  ,
        R[5]TeX.ams     ,
        R[7]TeX.bibtex
      }
  }
```

3 Examples

In the following examples, for simplicity, mostly LATEX 2_ε commands are used. Furthermore, they are based on the commands from this package and the ones from *codeverbatim* [2], since this package alone has no provision for grabbing verbatim code.

Moreover, for simplicity (and keeping them in sync) this section is duplicated in both packages, resulting in both packages being cited, `\cite{}`, instead of using internal links, `\ref{}`.

3.1 Preloading sets

In the preamble (see *"Pre-defined sets"* from *codehighlight* [1]):

```
% only in the preamble
\hlLangLoad{ C , php , TeX }
```

This will trigger the loading of the files associated with *C*, *php* and *TeX*. Note that, since *php* is based on *html* the file associated with it will also be loaded.

> ***Warning:*** Keep in mind that loading of files can only happen in the preamble, otherwise an error will ensue.

Note: This isn't mandatory if, in the preamble, one uses `\hlSettings`, referencing the desired language/set or `\hlLangDef` / `\hlKeysetDef`, either extending or based on the desired language/set. Those commands also trigger the loading of the needed auxiliary files.

3.2 A minimal construct

A minimal construct could entice setting only keywords (but no language, therefore, no comment or string will be recognized), and define an environment which will display the collected verbatim code right away (see *"Display Settings"* from *codehighlight* [1] and *"New Verbatim Environment"* from *codeverbatim* [2]):

```
% if the file definition were already loaded, the following can be anywhere
\hlSettings
  {
    keywd set = { C }
  }

\cvNewEnvironment[verb tl=\DemoV]{DemoEnv}{}
  {}
  {
  \hlDisplayV \DemoV
%  \myShow \DemoV
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment
```

Changing the settings to include the language syntax:

```
\hlSettings
  {
    language = C ,
    keywd set = C
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment
```

3.3 Customizing the language

One can change an existing language definition, for instance, adding a LaTeX escape (the *nestable* key is to allow it to be detected even inside another comment, see *"Language Syntax"* from *codehighlight* [1]):

```
\hlLangDef{C}
  {
    active comment line = {1}{\$\$}{ nestable }
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment $$ \textbf{this:} $A_i$
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment this: A_i
```

3.4 Further customization

One can use color to call attention to some parts of the code, in general this is achieved by customizing a ⟨hlset⟩ (see *"Highlight Sets"* from *codehighlight* [1]). Note that, in the following example, ⟨std-hlset⟩ is a pre-defined ⟨hlset⟩ (starting one).

```
\hlLangDef{C}
  {
    comment line = {2}{//+}{ }
  }
\hlHLsetDef{std-hlset}
  {
    comment = {2}{\itshape\color{red}}
  }
```

```
  \begin{DemoEnv}
    for ( i=0 , i<10 , i++ ) {
      g += i;
    }
    // a comment $$ \textbf{this:} $A_i$
    //+ really important comment
  \end{DemoEnv}
```

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}
// a comment this:  A_i
//+ really important comment
```

3.5 Using sequences

Note, the following example could be written in LaTeX 2ε, but, it's more natural to write it in *expl3* [6] (given the use of sequences). First a dedicated store command is created (see *"Store commands"* from *codeverbatim* [2]) then an environment using it (see *"New Verbatim Environment"* from *codeverbatim* [2]).

```
\ExplSyntaxOn

  \codeverb_new_store_cmds:nn { } { DemoStore }
  \cs_generate_variant:Nn \DemoStore_gput_right:nn { V }

  \codeverb_new_environment:nnnnn
    {
      drop ~ first,
      inline ,       % The environment end code will receive the verbatim code as #1
      other =  { \^^K } ,
    }
    {DemoStore} {O{default}}
    { %% Here, #1 is the environment optional parameter
      \tl_set:Nn \DemoStore_tmp_tl {#1}
    }
    { %% Here, #1 is the collected verbatim code
      \DemoStore_gput_right:Vn
          \DemoStore_tmp_tl
          { #1 }
    }
\ExplSyntaxOff
```

The code above defines a new environment, *DemoStore*, which expects a single optional parameter: the 'sequence name' to be used. Once a few verbatim snips are collected, they can be merged (command `\DemoStoreMerge` below, created by the command `\codeverb_new_store_cmds:nn` above) and then displayed as a simple piece.

```
\begin{DemoStore}[sampleA]
      for ( i=0 , i<10 , i++ ) {
      g += i;
    }    // a comment $$ \textbf{this:} $A_i$
\end{DemoStore}

\begin{DemoStore}[sampleA]
      for ( j=0 , j<10 , j++ ) {
      h += i;
    }    // a comment $$ \textbf{this too:} $B_j$
\end{DemoStore}

\DemoStoreMerge {\mymerged} {sampleA} {1-2}
\hlDisplayV \mymerged
```

```
  for ( i=0 , i<10 , i++ ) {
  g += i;
}    // a comment this:  A_i
  for ( j=0 , j<10 , j++ ) {
  h += i;
}    // a comment this too:  B_j
```

3.6 Verbatim shortcuts

The following is to be used with short verbatim commands (see *"Short Verbatim"* from `codeverbatim` [2]).

```
\hlBasefmtDef { ShortVerb }  %%% all non given keys start with their default values
  { inline }
```

3.6.1 Custom settings

A few dedicated settings, per language:

```
\hlSettingsDef { PHP }
  {
    based on language    = { php }  ,
    based on keywd set   = { html , php }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { ShortVerb }
  }
```

```
\hlSettingsDef { C++ }
  {
    based on language    = { C++ }  ,
    based on keywd set   = { C , C++ }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { std-basefmt }
  }
```

```
\hlSettingsDef { C }
  {
    based on language    = { C }  ,
    based on keywd set   = { C }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { std-basefmt }
  }
```

```
\hlSettingsDef { TeX }
  {
    based on language    = { TeX }  ,
    based on keywd set   = { TeX.common }  ,
    based on hl set      = { std-hlset }  ,
    based on base format = { ShortVerb }
  }
```

Note that, all sets have their own "name space", You can have a settings named "C++", a language named "C++", a keyword set named "C++" and so own. None correlated, or attached, to the other.

3.6.2 Custom highlight set

Besides being an example of a custom highlight set, this is just to emphasize the use of the shortcuts commands in the following examples.

```
\hlHLsetDef { std-hlset }
  {
   keywd  = {1}{\bfseries\color{blue}} ,
   php    = {1}{\bfseries\color{red}}  ,
   html   = {1}{\bfseries\color{green}}  ,
   string = {2}{\slshape\color{white!50!red}}  ,
   string = {1}{\slshape\color{white!50!black}}  ,
  }
```

3.6.3 Custom shortcuts

The following will define 4 shortcuts, one "raw" (showing spaces), one for TEX commands, one for C++ and another one for html/php (see *"Short Verbatim"* from `codeverbatim` [2]).

```
\cvSVerbNew [no endmark]{|}{|}
  {#1}

\cvSVerbNew {||}{||}
  {\hlSettingsSelect{TeX}\hlDisplay[inline]{#1}}

\cvSVerbNew {|<}{>|}
  {\hlSettingsSelect{PHP}\hlDisplay{#1}}

\cvSVerbNew {|(}{)|}
  {\hlSettingsSelect{C++}\hlDisplay[inline]{#1}}
```

Note that, in the first case, since the collected code will be used "raw" (no processing), the *no endmark* is used to suppress the insertion of an end mark, `^^J`, which would, otherwise, be rendered as an small speck, like ˙.

> ***Note:*** If the key *inline* is part of the base format, it is possible to omit it when using `\hlDisplay`.

Using an explicit short verbatim command

```
%% this will set it, plain verbatim text, hidding spaces.
\cvSVerbSet[no  endmark]{\cvHideSpc{#1}}

then you can have \cvSVerb!a \short \one ...! in the middle of some text.
```

then you can have a \short \one ... in the middle of some text.

Or a more elaborated example of use, turning on the short verbatim shortcuts, `\cvSVerbOn`

```
\cvSVerbOn
  A more elaborated case,
  showing spaces,  |quick \and \simple| or
  some \LaTeX{} source code like || this is \TeX{} || followed by
  simple php |< <?php echo "some text"; ?> some other echo >|,
  why not C like |( for ( j=0 , j<10, j++) )| or
  some php inside some html |< <a source="<?php echo "some text"; ?>" > and    >|.
  All in the same paragraph.

\cvSVerbOff
```

A more elaborated case, showing spaces, quick␣\and␣\simple or some LaTeX source code like `this is \TeX{}` followed by simple php `<?php echo "some text"; ?> some other echo` , why not C like `for ( j=0 , j<10, j++)` or some php inside some html `<a source="<?php echo "some text"; ?>" > and  ` . All in the same paragraph.

> ***Note:*** When using these shortcuts, take note that `\peek_regex:NTF` is used to recognize the many commands that start with the same (active) character, which has the side effect that the first following character will be captured with the current code régime. See *"Short Verbatim"* and *"caveats"* from *codeverbatim* [2] for details.

3.7 A floating environment

Using the package *newfloat* [5], one can declare a new floating environment (including a "list of ...") as follow:

```
%% This must be declared in the preamble
\DeclareFloatingEnvironment
  [fileext={lov},listname={List of C code},name={C code},placement=htb]
  {floatenv}
```

Then, it's a matter of declaring a dedicated verbatim environment that uses the floating environment declared above:

```
\ExplSyntaxOn
  \codeverb_new_environment:nnnnn
    {
      drop ~ first,
      inline ,       % The environment end code will receive the verbatim code as #1
    }
    {DemoFloat} {m}
    { %% Here, #1 is the environment optional parameter
      \tl_set:Nn \l_caption_tl {#1}
    }
    { %% Here, #1 is the collected verbatim code
      \begin{floatenv}
        \exp_args:No \caption \l_caption_tl
        \hlSettingsSelect{C}
        \hlDisplay{#1}
      \end{floatenv}
    }
\ExplSyntaxOff
```

Lastly, one can just use it as any other floating environment:

```
\begin{DemoFloat}{C sample}
  for ( i=0 , i<10 , i++ ) {
    g += i;
  }    // a detail $$ $B_j$
\end{DemoFloat}
```

C code 1: C sample

```
for ( i=0 , i<10 , i++ ) {
  g += i;
}    // a detail B_j
```

3.8 Using listings with codeverbatim

Another possibility, is using *codeverbatim* [2] together with *listings* [4], that for is enough to set a *listings* environment (see it's manual for details), a "retokenizing" command to re-scan the verbatim code with *listings*, `\cvRetokenEnvNewCommand` below, see *"Retokenizing"* from *codeverbatim* [2], and a verbatim environment which uses the "retokenizing" command:

```
\lstnewenvironment{cvlst}
  { \lstset{language=Pascal} }
  {}

\cvRetokenEnvNewCommand
  {cvlisting} {cvlst}

  \cvNewEnvironment
    [
      drop first = false,
      inline ,       % The environment end code will receive the verbatim code as #1
      other =  { \^^K } ,
    ]
    {DemoListing} {}
    { %% Here, #1 is the environment optional parameter
    }
    { %% Here, #1 is the collected verbatim code
      %% note the preceeding ^^J
      \cvlisting {^^J #1}
    }
```

Note: The `^^J` is to "give" listings a first empty line, so it won't complain about it.

Thereafter, this is like any other listing environment, though, one where one could further customize every aspect of it, like preprocessing the code, using sequences to store the code before retokenizing it, etc..

```
\begin{DemoListing}
 for i:=maxint to 0 do
 begin
 { do nothing }
 end;

 Write('Case insensitive ');
 WritE('Pascal keywords.');
\end{DemoListing}
```

```
 for i:=maxint to 0 do
begin
{ do nothing }
end;

Write('Case insensitive ');
WritE('Pascal keywords.');
```

4 To do, Future plans

These are some aspects (and random ideas) that are either planned or that deserve some consideration:

- grabbing code from an external file (codeverbatim): defining a 'text marker' (begging/end ... label).
- optimize the background color code. It works as is, but, it's slow.
- "here code" typical syntax <<Label or <<'Label'. Possible approach: _scanT collects the next label and sets _end_regex.
- handling of tabs (? fixed spacing or column mod num ?).
- 'simple' identifiers :
  - (maybe) <start>... followed by an explicit regex (expl3): not really an identifier (name it 'markers' ??) like <start> => `#` <regex> `\d \d \d | \d \d | \d` (that would make for `#1` or `#20` or `#250` (Delphi)). That could also be used to highlight 'other keywords' (if they fit a regular expression).
- 'delimited' identifiers :
  - <start> ... <end> (some multi-word compiler directives). code problem: a 'keyword' may contain an space then. phrase keywords.
  - (maybe) delimited identifier followed by regex (for instance, *expl3* signatures). issue: needs a different/unique ⟨start-pattern⟩ (no backtrack).
- (maybe) blank lines at the end (codeverbatim?)... possible, but would need some temp (seq? tl? int?) holding 'blank lines' (delayed insertion) ... waiting to see if something else comes along...main problem: user supplied 'line processing'... before or after it?
- indexing might be implement in a similar way as it is in *codedescribe*.
- allow for further customization of line numbering: e.g. one every 5, or starting number, etc.
- (maybe) change the markers from `\q_nil` to something more end user safe, like `\s_stop` or `\scan_stop:` ("drawback" no pre-existent cmd like `\quark_if_nil:`... but could prevent a quark loop in case of misuse.)
- simula/algol comments peculiar syntax.
- listings *procnamekeys* might be implemented. Idea: add another 'list' (like identifiers) whose 'result' would be setting a flag for the following keyword.
- writing code to an external file, albeit fairly simple, isn't planned for the time being.
- improve the examples section.

References

[1] Alceu Frigeri. *The codehighlight Package*. 2026, p. 24. URL: https://mirrors.ctan.org/macros/latex/contrib/codehighlight/doc/codehighlight.pdf (visited on 11/30/2026).

[2] Alceu Frigeri. *The codeverbatim Package*. 2026, p. 14. URL: https://mirrors.ctan.org/macros/latex/contrib/codeverbatim/doc/codeverbatim.pdf (visited on 11/30/2026).

[3] Pablo González. *SCONTENTS - Stores LaTeX Contents*. 2024, p. 48. URL: http://mirrors.ctan.org/macros/latex/contrib/scontents/scontents.pdf (visited on 03/10/2025).

[4] Jobst Hoffmann. *The Listings Package*. 2024, p. 65. URL: http://mirrors.ctan.org/macros/latex/contrib/listings/listings.pdf (visited on 03/10/2025).

[5] Axel Sommerfeldt. *The newfloat package*. 2023, p. 8. URL: https://mirrors.ctan.org/macros/latex/contrib/newfloat/newfloat.pdf (visited on 11/30/2026).

[6] The LaTeX Team. *The LaTeX3 Interfaces. The LaTeX Project*. 2026, p. 385. URL: https://mirrors.ctan.org/macros/latex/required/l3kernel/interface3.pdf (visited on 08/07/2026).

[7] Timothy Van Zandt. *The 'fancyvrb' package. Fancy Verbatims in LATEX*. 2026, p. 24. URL: http://mirrors.ctan.org/macros/latex/contrib/fancyvrb/doc/fancyvrb-doc.pdf (visited on 06/19/2026).