

Common Lisp Documentation Weaver

Mariano Montone (marianomontone@gmail.com)

Copyright © 2021 Mariano Montone

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, with no Front-Cover Texts, and with no Back-Cover Texts. A copy of the license is included in the section entitled “GNU Free Documentation License”.

Table of Contents

1	Introduction	1
2	Installation	2
3	Usage	3
3.1	Command Line	3
4	Commands	5
5	Documentation systems	8
5.1	Texinfo	8
5.1.1	common-lisp.texi	8
6	Tips and tricks	11
6.1	Lisp evaluation	11
7	API	14
8	Index	16

1 Introduction

CL-DOCWEAVER is a document weaver for Common Lisp.

Documentation for a Lisp project is written with the user's tool of choice (like Texinfo, Markdown, etc). Then, Common Lisp definitions are expanded into the documentation source using DocWeaver commands.

DocWeaver commands give the user control on how definitions are to be expanded, either via command options or by choosing a different set of commands.

CL-DOCWEAVER is easy to extend to support different documentation tools.

Texinfo and Markdown are the ones with best support at this moment.

2 Installation

3 Usage

Write documentation for your Common Lisp project in your documentation tool of your choice (either Texinfo or Markdown at this moment). Then invoke *cl-docweaver* commands to expand Lisp definitions for either variables, functions, macros, classes, or even whole packages.

Commands have the following syntax: (*@command-name* &rest *args*).

For example, use (*@clfunction alexandria:flatten*) to expand the definition of *ALEXANDRIA:FLATTEN* function.

The expanded function definition looks like this:

```
ALEXANDRIA:FLATTEN (tree) [Function]
  Traverses the tree in order, collecting non-null leaves into a list.
```

Note that commands usually receive options in order to be able to control different aspects of the expanded definition.

By default, docstrings are interpreted to extract possible references to other parts of the code; then those references are formatted as links that can be used to navigate the definitions documentations.

Finally, use [WEAVE-FILE], page 14 to weave your documentation system source files.

Have a look at *cl-docweaver* documentation in *docs* directory for an example of how all this works.

3.1 Command Line

The *cl-docweaver* executable can be used weave documents from the command line.

To build, do:

```
make
sudo make install
```

Then use it like this: T

```
cl-docweaver - Common Lisp Documentation Weaver
```

```
USAGE: sbcl [OPTIONS] INPUTFILE
```

```
Weave Common Lisp documentation in INPUTFILE.
```

Options:

```
--version          display version information and exit
-h, --help         display help information and exit
-d, --debug        debug
-o FILE, --output FILE
                    output file
-s DOCSYSTEM, --docsystem DOCSYSTEM
                    the documentation system to use. either texinfo or
                    markdown. if not specified, the documentation system
                    used is inferred by looking at input file extension.
```

```

-m MODULES, --modules MODULES
                        the list of modules to REQUIRE
-c COMMAND-PREFIX, --command-prefix COMMAND-PREFIX
                        the command prefix character to use. Default is --parse-doc
                        enabled by default.
--escape-docstrings    When enabled, escape the docstrings depending on the
                        output. This is enabled by default.

```

Examples:

Weave texinfo file and visualize weaved output

```
cl-docweaver my-documentation.texi
```

Weave texinfo file into a file

```
cl-docweaver my-documentation.texi -o my-documentation.weaved.texi
```

4 Commands

@setup &rest options [Command]

Configures *cl-docweaver*.

OPTIONS is a *plist* with members:

- **:docsystem** The documentation system to use. Either **:texinfo** or **:markdown**. Default is **:texinfo**.
- **:parse-docstrings** A boolean that indicates if docstrings should be parsed or not. Default is T.
- **:command-prefix** The prefix character to use for commands. Default is the **#\@** character.
- **:modules** A list of Lisp modules to **REQUIRE** in order to be able to expand definitions.

@clvariable variable-symbol &rest args [Command]

Expands definition for variable bound to *VARIABLE-SYMBOL*.

For example,

```
(@clvariable cl:*standard-output*)
```

Expands to this:

```
*STANDARD-OUTPUT* [COMMON-LISP]
  default output stream
```

A list of symbols is also accepted; variable definitions are expanded in sequence.

For example,

```
(@clvariable (cl:*compile-print* cl:*compile-verbose*))
```

expands to this:

```
*COMPILE-PRINT* [COMMON-LISP]
  The default for the :PRINT argument to [COMPILE-FILE], page 12.
```

```
*COMPILE-VERBOSE* [COMMON-LISP]
  The default for the :VERBOSE argument to [COMPILE-FILE], page 12.
```

@clfunction function-symbol &rest args [Command]

Expands definition for function bound to *FUNCTION-SYMBOL*.

For example,

```
(@clfunction alexandria:map-permutations)
```

Expands to this:

```
ALEXANDRIA:MAP-PERMUTATIONS (function sequence &key (start 0) end length (copy t)) [Function]
```

Calls *function* with each permutation of *LENGTH* constructable from the subsequence of *SEQUENCE* delimited by *START* and *END*. *START* defaults to 0, *END* to *length* of the *sequence*, and *LENGTH* to the *length* of the delimited subsequence.

Like with See `<undefined>` [`@clvariable`], page `<undefined>`, a list of symbols is also accepted and definitions are expanded in sequence.

@clmacro *macro-symbol* **&rest** *args* [Command]

Expands definition for macro bound to *MACRO-SYMBOL*.

For example,

```
(@clmacro cl:print-unreadable-object)
```

Expands to this:

COMMON-LISP:PRINT-UNREADABLE-OBJECT ((*sb-impl::object* [Macro]
stream **&key** *type identity*) **&body** *sb-impl::body*)

Output *OBJECT* to *STREAM* with "#`<undefined>` [`<`], page `<undefined>`" prefix, "`<undefined>` [`>`], page `<undefined>`" suffix, optionally with object-type prefix and object-identity suffix, and executing the code in *BODY* to provide possible further output.

As in other commands, a list of symbols is also accepted; macro definitions are expanded in sequence.

@clclass *class-name* **&rest** *args* [Command]

Expands definition of class with name *CLASS-NAME*.

For example,

```
(@clclass asdf:component)
```

Expands to this:

ASDF/COMPONENT:COMPONENT [Class]

Base class for all components of a build

Class precedence list: `component`, `standard-object`, `t`

Slots:

- **name** — initarg: `:name`; reader: `asdf/component:component-name`; writer: `(setf asdf/component:component-name)`

Component name: designator for a string composed of portable pathname characters

As in other commands, a list of symbols is also accepted; class definitions are expanded in sequence.

@clpackage *package-name* **&key** (*include-external-definitions* **t**) [Command]
include-internal-definitions (*categorized* **t**)

Expands definition for Common Lisp package named *PACKAGE-NAME*.

If *INCLUDE-EXTERNAL-DEFINITIONS* is **T**, then all package external definitions are expanded.

If *INCLUDE-INTERNAL-DEFINITIONS* is **T**, then all package internal definitions are expanded.

CATEGORIZED controls how to categorize the expanded package definitions:

- *:by-kind* or **T**, definitions are separated in sections (variables, functions, etc).

- *:by-docstring-category*, definitions are grouped by the category parsed from docstrings. A category for a definition is specified by adding the text “Category: *category-name*” to the docstring.
- Otherwise, they are expanded in sequence with no separation.

Example:

```
(@clpackage :alexandria)
```

@clref *symbol type* [Command]

Creates a reference to *SYMBOL*. *TYPE* should be one of **variable**, **function**, **class**, etc.

For example, to reference ALEXANDRIA:FLATTEN function, do this:

```
(@clref alexandria:flatten function)
```

And this is the resulting link: [FLATTEN], page 3

@cleva *expression* [Command]

Evaluates Lisp *EXPRESSION* and prints the result.

@clcapture-output *expression* [Command]

Evaluates Lisp *EXPRESSION*, captures its output, and prints it to the document.

5 Documentation systems

5.1 Texinfo

The Texinfo output needs to include `common-lisp.texi` file, that is shipped with *CL-DOCWEAVER*.

The `common-lisp.texi` file contains a set of Texinfo macros that are used by *CL-DOCWEAVER* for expanding Common Lisp definitions.

You can have a look at *CL-DOCWEAVER* own documentation in `docs/cl-docweaver.texi` for an example for how this should be used.

Also you may want to invoke `makeinfo` and `texi2any` Texinfo commands with `--no-validate` option, as some of the generated references in docstrings may not appear in your final document, and without that option you would get an error.

See `docs/Makefile` in *CL-DOCWEAVER* source for an example of how Texinfo tools should be used.

5.1.1 common-lisp.texi

`common-lisp.texi` file contains macros for defining Common Lisp related definitions.

They are mostly equivalent to Texinfo's definition macros, like `@defn`, `@defun`, etc, but for Common Lisp. In particular, they take into consideration Lisp packages, and uses them for naming and index entries.

`@cldefun` is for defining a Common Lisp function.

They are used like this:

```
@cldefun{alexandria, flatten, ()}
  Traverses the @var{tree} in order, collecting non-null leaves into a list.
@endcldefun
```

You can use the macros in `common-lisp.texi` to define your own Common Lisp definitions manually, without using *CL-DOCWEAVER* expanders.

Have a look at the source to figure out more about how they are used:

`@c Macros for Common Lisp definitions`

```
@c Variable definition
@macro cldefvar{package, name}
@vindex \package\:\name\
@anchor{\package\:\name\ variable}
@defvr \package\ \name\
@end macro
```

```
@macro endcldefvar
@end defvr
@end macro
```

```
@c Function definition
@macro cldefun{package, name, args}
```

```

@findex \package\:\name\
@anchor{\package\:\name\ function}
@defun \package\:\name\ \args\
@end macro

@macro endcldefun
@end defun
@end macro

@c Example:
@c @cldefun {alexandria, flatten, (x y z)}
@c This is alexandria flatten function
@c @endcldefun

@c Function definition
@macro cldefmacro{package, name, args}
@findex \package\:\name\
@anchor{\package\:\name\ macro}
@defmac \package\:\name\ \args\
@end macro

@macro endcldefmacro
@end defmac
@end macro

@c Example:
@c @cldefmacro {alexandria, with-gensyms, (&rest args)}
@c This is alexandria with-gensyms macro
@c @endcldefmacro

@c Generic function definition
@macro cldefgeneric{package, name, args}
@findex \package\:\name\
@anchor{\package\:\name\ function}
@defn Generic-Function \package\:\name\ \args\
@end macro

@macro endcldefgeneric
@end defn
@end macro

@c Class definition
@macro cldefclass{package, name}
@tindex \package\:\name\
@anchor{\package\:\name\ class}
@deftp Class \package\:\name\
@end macro

```

```
@macro endcldefclass
@end deftp
@end macro
```

```
@c References
@macro clref{package, name, type}
@ref{\package\:\name\ \type\,\name\, \name\}
@end macro
```

```
@c Source references
@macro clsourceref{type,package,name}
@end macro
```

```
@c Use @clref{package, name} to reference cl definitions
```

```
@c Weave Common Lisp function definition
@macro clfunction{package, name}
@end macro
```

```
@macro clsourcecode{system,path}
@end macro
```

```
@macro setup{things}
@end macro
```

The `common-lisp.texi` file is required to be included in the file being weaved by CL-DOCWEAVER for the Texinfo documentation system, as the implementation expands to macros found in `common-lisp.texi`.

6 Tips and tricks

6.1 Lisp evaluation

It is possible to take advantage of Lisp evaluation to handle the list of symbols to expand. As commands are parsed using standard `CL:READ` function, reader syntax `#.` can be used to evaluate arbitrary Lisp code.

COMMON-LISP:READ (**&optional** (*stream* **standard-input**) [Function]
 (*sb-impl::eof-error-p* *t*) (*sb-impl::eof-value* *nil*) (*sb-impl::recursive-p* *nil*))
 Read the next Lisp value from *STREAM*, and return it.

We can take advantage of that and expand all functions that match some term.

Symbols matching

For example, to expand all functions in `CL` package that have 'file' in their name:

```
(@clfunction #.(docweaver/utils:symbols-matching :cl "FILE" :function))
```

Results in this expansion:

COMMON-LISP:PROBE-FILE (*sb-impl::pathspec*) [Function]
 Return the truename of *PATHSPEC* if the truename can be found,
 or `NIL` otherwise. See `<undefined>` [TRUENAME], page `<undefined>` for more information.

COMMON-LISP:FILE-AUTHOR (*sb-impl::pathspec*) [Function]
 Return the author of the file specified by *PATHSPEC*. Signal an
 error of type if no such file exists, or if *PATHSPEC*
 is a wild pathname.

COMMON-LISP:RENAME-FILE (*sb-impl::file* *sb-impl::new-name*) [Function]
 Rename *FILE* to have the specified *NEW-NAME*. If *FILE* is a stream open to a
file, then the associated *file* is renamed.

COMMON-LISP:FILE-LENGTH (*stream*) [Function]

COMMON-LISP:FILE-POSITION (*stream* **&optional** *position*) [Function]

COMMON-LISP:FILE-NAMESTRING (*pathname*) [Function]
 Return a string representation of the name in *PATHNAME*.

COMMON-LISP:DELETE-FILE (*sb-impl::file*) [Function]
 Delete the specified *FILE*.

If *FILE* is a stream, on Windows the stream is closed immediately. On Unix platforms the stream remains open, allowing IO to continue: the OS resources associated with the deleted *file* remain available till the stream is closed as per standard Unix `unlink()` behaviour.

COMMON-LISP:COMPILE-FILE-PATHNAME (*sb-c::input-file* **&key** [Function]
(sb-c::output-file nil sb-c::output-file-p) &allow-other-keys)

Return a pathname describing what file [COMPILE-FILE], page 12 would write to given these arguments.

COMMON-LISP:COMPILE-FILE (*sb-c::input-file* **&key** (*sb-c::output-file* [Function]
(sb-c::cfp-output-file-default sb-c::input-file)) (*(:verbose*
**compile-verbose*) *compile-verbose*)* (*(:print *compile-print*)*
**compile-print*)* (*sb-c::external-format :default*) (*sb-c::trace-file nil*)
*((:block-compile sb-c::*block-compile-arg*) nil) (sb-c::emit-cfasl*
*sb-c::*emit-cfasl*))*

Compile *INPUT-FILE*, producing a corresponding fasl file and returning its filename.

:PRINT

If true, a message per non-macroexpanded top level form is printed to [*STANDARD-OUTPUT*], page 5. Top level forms that whose subforms are processed as top level forms (eg. *<undefined>* [EVAL-WHEN], page *<undefined>*, *<undefined>* [MACROLET], page *<undefined>*, *<undefined>* [PROGN], page *<undefined>*) receive no such message, but their subforms do.

As an extension to ANSI, if *:PRINT* is *:top-level-forms*, a message per top level form after macroexpansion is printed to [*STANDARD-OUTPUT*], page 5.

For example, compiling an *<undefined>* [IN-PACKAGE], page *<undefined>* form will result in a message about a top level *<undefined>* [SETQ], page *<undefined>* in addition to the message about the *<undefined>* [IN-PACKAGE], page *<undefined>* form' itself.

Both forms of reporting obey the *<undefined>* [SB-EXT:*COMPILER-PRINT-VARIABLE-ALIST*], page *<undefined>*.

:BLOCK-COMPILE

Though [COMPILE-FILE], page 12 accepts an additional *:BLOCK-COMPILE* argument, it is not currently supported. (non-standard)

:TRACE-FILE

If given, internal data structures are dumped to the specified file, or if a value of is given, to a file of *<undefined>* [*], page *<undefined>*.trace type derived from the input file name. (non-standard)

:EMIT-CFASL

(Experimental). If true, outputs the toplevel compile-time effects of this file into a separate .cfasl file.

`COMMON-LISP:FILE-ERROR-PATHNAME` (*condition*) [Function]

`COMMON-LISP:FILE-WRITE-DATE` (*sb-impl::pathspec*) [Function]

Return the write date of the file specified by *PATHSPEC*.

An error of type is signaled if no such file exists,
or if *PATHSPEC* is a wild pathname.

`COMMON-LISP:FILE-STRING-LENGTH` (*stream sb-impl::object*) [Function]

Categorized definitions

We can also go a bit further and use evaluation to fetch a list of symbols with docstrings matching a certain category. When the syntax `Category: <category-name>` is used in definition docstrings, it is detected by See `<undefined>` [docweaver/utls:symbols-categorized], page `<undefined>`.

For example:

```
(@clfunction #.(docweaver/utls:symbols-categorized :docweaver/utls
"foobar" :function))
```

Expands the categorized functions:

`DOCWEAVER/UTILS:BAR` *nil* [Function]

[BAR], page 13 function.

Category: foobar.

`DOCWEAVER/UTILS:FOO` *nil* [Function]

[FOO], page 13 function.

Category: foobar.

7 API

DOCWEAVER

[PACKAGE]

External definitions

Macros

DOCWEAVER:DEF-WEAVER-COMMAND-HANDLER (*command-name args* [Macro]
(**&key** *docsystem*) **&body** *body*)

Define a weaver command handler.

COMMAND-NAME is the name of the command, without the prefix (like 'clvariable', 'clfunction', etc.)

ARGS is the list of arguments for that command in the *DOCSYSTEM* implementation.

DOCSYSTEM is a specializer for the documentation system. For example, (eql *:texinfo*).

BODY should write to an implicit variable, to expand the command.

This is implemented as a wrapper over [PROCESS-WEAVER-COMMAND], page 14 .

Generic functions

DOCWEAVER:PROCESS-WEAVER-COMMAND (*docsystem command* [Generic-Function]
args stream)

The generic function to specialize for implementing weaving commands for the different documentation systems.

See: <undefined> [DEF-WEAVER-COMMAND-HANDLER], page <undefined>

Functions

DOCWEAVER:WEAVE-FILE (*file output-file &rest options &key* [Function]
docsystem modules command-prefix (parse-docstrings t)
(escape-docstrings t))

Weaves documentation source in *FILE* and writes the result to *OUTPUT-FILE*.

Arguments:

- *DOCSYSTEM* : specify the documentation tool that is being used (*:texinfo*, *:markdown*, etc.).
- *MODULES* : is the list of *modules* (or ASDF system names) that need to be loaded to be able to read definition descriptions.
- *COMMAND-PREFIX* : is the character to use as prefix for commands. The character 'at' is the default.
- *PARSE-DOCSTRINGS* : if *T*, then docstrings are parsed and highlighted and references to code from it created.

- `ESCAPE-DOCSTRINGS`: if `T`, then docstrings are escaped by the documentation system. Escaping allows the use of special documentation system characters in docstring sources. If the escaping of docstrings is turned off, then that allows to use documentation system markup in docstrings.

Category: `TopLevel`

8 Index

(Index is nonexistent)

*

COMPILE-PRINT	5
COMPILE-VERBOSE	5
STANDARD-OUTPUT	5

@

@clcapture-output	7
@clclass	6
@cleval	7
@clfunction	5
@clmacro	6
@clpackage	6
@clref	7
@clvariable	5
@setup	5

A

ALEXANDRIA:FLATTEN	3
ALEXANDRIA:MAP-PERMUTATIONS	5

C

COMMON-LISP:*COMPILE-PRINT*	5
COMMON-LISP:*COMPILE-VERBOSE*	5
COMMON-LISP:*STANDARD-OUTPUT*	5

C

COMMON-LISP:COMPILE-FILE	12
COMMON-LISP:COMPILE-FILE-PATHNAME	12
COMMON-LISP:DELETE-FILE	11
COMMON-LISP:FILE-AUTHOR	11
COMMON-LISP:FILE-ERROR-PATHNAME	13
COMMON-LISP:FILE-LENGTH	11
COMMON-LISP:FILE-NAMESTRING	11
COMMON-LISP:FILE-POSITION	11
COMMON-LISP:FILE-STRING-LENGTH	13
COMMON-LISP:FILE-WRITE-DATE	13
COMMON-LISP:PRINT-UNREADABLE-OBJECT	6
COMMON-LISP:PROBE-FILE	11
COMMON-LISP:READ	11
COMMON-LISP:RENAME-FILE	11

D

DOCWEAVER/UTILS:BAR	13
DOCWEAVER/UTILS:FOO	13
DOCWEAVER:DEF-WEAVER-COMMAND-HANDLER	14
DOCWEAVER:PROCESS-WEAVER-COMMAND	14
DOCWEAVER:WEAVE-FILE	14