The Declt User Manual

Documentation Extractor from Common Lisp to Texinfo, Version 4.0 beta 2 "William Riker"

Didier Verna <didier@didierverna.net>

Table of Contents

Copying

1 Introduction

`Declt` (pronounce "dec'let") is a reference manual generator for Common Lisp libraries. A `Declt` manual documents one specified ASDF system (considered as the "main" system), and all its local dependencies (subsystems found in the same distribution). This is what is collectively referred to as the *library*.

`Declt` doesn't perform any kind of static code analysis, but instead loads the library, and then introspects the Lisp environment to discover what "belongs" to it. The generated documentation includes the description of both programmatic and ASDF components. Every such component description is called a *definition*.

`Declt` manuals provide a detailed description of the library's infrastructure by including definitions for every relevant ASDF component (systems, modules, and files), and Lisp package.

Exported programmatic definitions are split from the internal ones, which allows to separately browse either the library's public interface or its implementation. Both sections of the manual include definitions for constants, special variables, symbol macros, macros, setf expanders, compiler macros, regular functions (including setf ones), generic functions and methods (including setf ones), method combinations, conditions, structures, classes, and types.

Programmatic definitions are as complete and exhaustive as introspection can make them. `Declt` collects documentation strings, lambda lists (including qualifiers and specializers where appropriate), slot definitions (including type information, allocation type, initialization arguments, *etc.*), definition sources, *etc.*

Every definition includes a full set of cross-references to related ones: ASDF component dependencies, parents, and children, classes direct methods, super- and sub-classes, slot readers and writers, setf expanders access and update functions, *etc.*

Finally, `Declt` produces exhaustive and multiple-entry indexes to all documented aspects of the library.

`Declt` manuals are generated in Texinfo format. From there it is possible to produce readable / printable output in Info, HTML, PDF, Postscript, *etc.*

The primary example of documentation generated by `Declt` is the `Declt` *Reference Manual*.

2 Installation

First of all, see `Declt`'s homepage (`http://www.lrde.epita.fr/~didier/software/lisp/typesetting.php#declt`) for tarballs, Git repository and online documentation. `Declt` is an ASDF 3 library, and currently works with SBCL only. If you download a `Declt` tarball, or clone the repository, you need to unpack somewhere in the ASDF source registry. Otherwise, `Declt` is also available via Quicklisp. See Appendix B [Supported Platforms], page 29, for more information on portability and dependencies.

In addition to the bare Lisp library, the `Declt` distribution offers documentation in the form of 2 different manuals: the User Manual (you are reading it) and the `Declt` *Reference Manual*. The latter is generated by `Declt` itself. If you want to compile the manuals by yourself, please follow the instructions below.

1. Edit `make/config.make` to your specific needs. In particular, you will see a number of external programs that are required in order to compile the manuals in there (see Appendix B [Supported Platforms], page 29).
2. Type `make`. By default, the documentation is built in Info, PDF, and HTML formats. If you want other formats (DVI and Postscript are available), type `make all-formats`. You can also type individually `make dvi` and/or `make ps` in order to get the corresponding format.
3. Type `make install` to install the documentation. If you have compiled the documentation in DVI and Postscript format, those will be installed as well.

The reference manual's Texinfo source is included in the distribution (and in the repository), although, as mentioned before, it is generated by `Declt` itself. Before compiling, it is possible to regenerate a local version of it with hyperlinks to your installation by typing `make localref`. If you ever need to regenerate the regular version, you can also type `make generate`.

Type `make uninstall` to uninstall the library.

3 Quick Start

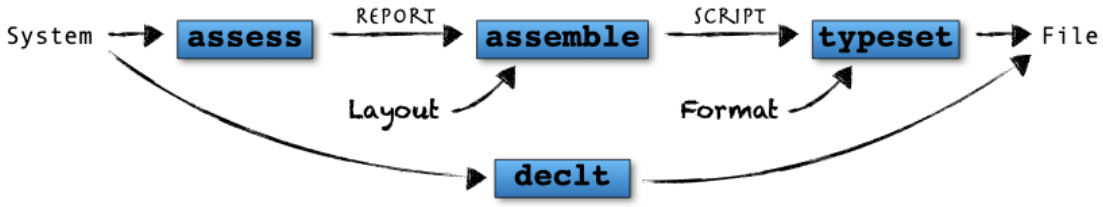
In your favorite Lisp REPL, type this.

```
(asdf:load-system :net.didierverna.declt) ;; or use ql:quickload
(net.didierverna.declt:nickname-package)
(declt:declt :my.asdf.system)
```

You will end up with a file named `my.asdf.system.texi` in the current directory. This is a Texinfo file that can further be compiled into various formats, such as Info, HTML, or PDF. For example, in order to get a PDF, type this in the same directory: '`makeinfo --pdf my.asdf.system.texi`', and enjoy reading the resulting `my.asdf.system.pdf`...

4 Overview

`Declt` is implemented as a 3 stages pipeline, as depicted in the figure below. The `declt` function, already mentioned in Chapter 3 [Quick Start], page 7, triggers the whole pipeline, but for more advanced usage, each stage of the pipeline can be accessed separately and directly via their respective entry point functions: `assess`, `assemble`, and `typeset`.



1. The first stage of the pipeline is called the *assessment* stage. At this stage, `Declt` loads the library, and then introspects the Lisp environment in order to extract the pertinent information. This information is stored in a so-called *report*. See Section 6.1 [Assessment], page 15, for more information.
2. The second stage of the pipeline is called the *assembly* stage. At this stage, `Declt` organizes the (completely flat) information provided by a report in a specific way. The result is called a *script*. A script begins to look like a properly organized reference manual, but it is still independent from the final output format. The specific organization scheme to use is called a *layout*. See Section 6.2 [Assembly], page 21, for more information.
3. Finally, the third stage of the pipeline is called the *typesetting* stage. At this stage, `Declt` renders a script to a file by typesetting its contents in a specific documentation format. This may directly result in a human-readable reference manual, although in the case of Texinfo, the resulting file needs additional post-processing. This, however, is not done by `Declt` itself, but by `makeinfo`, which is an external tool. See Section 6.3 [Typesetting], page 21, for more information.

Again, when you call `(declt system)`, the 3 stages of the pipeline are chained automatically, which is more or less equivalent to calling `(typeset (assemble (assess system)))` (see Chapter 5 [Global Usage], page 11).

5 Global Usage

This chapter explains how to use `Declt` to trigger the whole pipeline (see Chapter 4 [Overview], page 9), that is, to go directly from a library to a reference manual.

`Declt`'s main system is called '`net.didierverna.declt`'. Depending on your installation, you may thus either `asdf:load-system`, or `ql:quickload` it in your Lisp image.

`Declt`'s main package is called `net.didierverna.declt`. You can automatically nickname this package with the following function.

`nickname-package` [*nickname*] [Function]
Add *NICKNAME* (`:declt` by default) to the `:net.didierverna.declt` package.

In order to trigger the whole pipeline, use the `declt` function, which currently generates a Texinfo file containing the reference manual. This file may in turn be compiled into a variety of human-readable formats with the `makeinfo` program.

`declt` *system-name :key value...* [Function]
Generate a reference manual in Texinfo format for ASDF *SYSTEM-NAME*.
SYSTEM-NAME is an ASDF system designator.

The `declt` function accepts a number of keyword arguments which affect the behavior of every stage of the pipeline (they are in fact passed along to the corresponding functions, which see). The following sections describe these keyword arguments.

5.1 Assessment Options

Most of the documented material is extracted automatically by `Declt`, although sometimes it is difficult to get it right, if at all. From time to time, you may also want to override `Declt`'s default choices. The following options provide control over the extraction mechanism and result. They are passed along to the `assess` function (see Section 6.1 [Assessment], page 15), along with the library's system designator.

`:introspection-level`
How hard `Declt` introspects the Lisp environment in order to find information. Possible values currently are `1` (the default) or `2`. The `Declt` introspection heuristic establishes different compromises between completeness and performance. A higher introspection level will result in a more complete documentation, at the expense of a (much) greater computation time. See Section 6.1.2 [Introspection Levels], page 16, for more information.

`:library-name`
The library name, used in the reference manual title and at some other places. It defaults to the system name, but you are encouraged to provide a more human-readable version (for example `Declt` rather than just `net.didierverna.declt`).

`:tagline` A tagline for the library, used in the reference manual subtitle, or `nil`. It defaults to the system's long name or description (one-liners only), but see Section A.1.1 [Taglines], page 27.

`:library-version`
The library version, used in the reference manual subtitle and at some other places, or `nil`. It defaults to the system version (one-liner only).

`:contact` The contact(s) for the library, or `nil`. The default is extracted from the system definition (the `maintainer` and `author` slots, possibly also the `mailto` one; one-liners only). You may provide a contact string, or a list of such. A contact string is of the form `"My Name <my@address>"`, both name and address being optional.

`:copyright-years`
: Copyright year(s) used to typeset copyright header lines, or `nil` to disable those lines. It defaults to the current year. You may use any kind of string here, such as `"2013"`, `"2010, 2011"`, `"2010--2013"` *etc.*

`:license`
: The library's license type (`nil` by default). This information is used to insert licensing information at several places in the manual. The possible non-`nil` values are: `:mit`, `:boost`, `:bsd`, `:gpl`, `:lgpl`, and `:ms-pl`. The corresponding license file headers are stored in the `*licenses*` parameter. Please ask if you need other kinds of licenses added to `Declt`. Note that this information is **not** currently extracted from the system's `license` slot, as this slot is not well defined.

`:introduction`
: A potential contents for an introductory chapter (none by default). In the future, this material may be extracted from a README file or such.

`:conclusion`
: A potential contents for a conclusive chapter (none by default).

Note that both the introduction and the conclusion may contain Texinfo directives (no post-processing will occur). All other textual material is considered raw text and will be properly escaped for Texinfo.

5.2 Assembly Options

Based on the report created by the first stage of the `Declt` pipeline, the exact contents of the upcoming reference manual can be further adjusted thanks to the following options. They are passed along to the `assemble` function (see Section 6.2 [Assembly], page 21), along with the library's report.

`:declt-notice`
: Controls the output of a small paragraph about automatic manual generation by `Declt`. Possible values are `nil` (meaning don't acknowledge `Declt`), `:short`, and `:long` (the default). A `:short` paragraph advertises `Declt` with its version number. A `:long` one also prints its release code name and the generation time. I would be grateful if you kept at least the short version in your manuals, as an acknowledgment for using `Declt`.

`:locations`
: Whether to hyperlink definitions to their locations. It defaults to `nil`. A value of `:file-system` means to create hyperlinks on the local machine, on which `Declt` is currently being run. These links being completely specific to the current installation, it is better to avoid this option if the reference manual is meant to be put online.

`:default-values`
: Whether to document standard / default values. The default is `nil`. Otherwise, `Declt` will explicitly mention properties that do not differ from their default or standard value, such as a `standard` method combination, an `:instance` slot allocation, *etc.*

`:foreign-definitions`
: Whether to include foreign definitions in the documentation. It defaults to `nil`, in which case foreign references are still advertised, but do not point to actual definitions (they are not "clickable"). Otherwise, only foreign definitions somehow related to the library are considered for inclusion, and they will most of the time be partial rather than exhaustive: only the bits relevant to the library are documented. For more information on foreign definitions, see Section 6.1.1 [Domesticity], page 15.

5.3 Typesetting Options

Finally, the typesetting of the script produced by the second stage of the `Declt` pipeline may be adjusted with the following options. They are passed along to the `typeset` function (see Section 6.3 [Typesetting], page 21), along with the library's script.

`:output-directory`
: The generated file's directory. It defaults to the current directory.

`:file-name`
: The base name of the generated file, sans extension. It defaults to the system name.

`:info-name`
: The base name of the Info file, sans extension. It defaults to *FILE-NAME*. This option is provided because the Info file name appears in the Texinfo source.

`:info-category`
: The category under which the documentation will be listed in Info browsers. This corresponds to the `@dircategory` command in Texinfo, and defaults to `"Common Lisp"`.

6 Pipeline Usage

This chapter provides additional information on how each stage of the `Declt` pipeline works, and how to use them separately.

6.1 Assessment

As mentioned before (see Chapter 4 [Overview], page 9), the first stage of the `Declt` pipeline, called the "assessment" stage, loads the library, and then introspects the Lisp environment in order to extract the pertinent information.

The assessment part of `Declt` is provided as a separate system called '`net.didierverna.declt.assess`', which you may load individually (`Declt`'s main system depends on it). It also comes in a separate package called '`net.didierverna.declt.assess`' (imported by `Declt`'s main package).

In order to assess a library, use the `assess` function.

`assess` *system-name :key value...* [Function]
Extract and return documentation information for ASDF *SYSTEM-NAME*.
SYSTEM-NAME is an ASDF system designator.

The complete list of keyword arguments, also recognized by the global `Declt` function, is described in Section 5.1 [Assessment Options], page 11. This function returns an instance of the `report` class. The following sections provide additional information on the assessment's operation, and some details on the `report` class.

6.1.1 Domesticity

Domesticity is the concept defining what "belongs" to the library. It ultimately affects what can appear in the reference manual, and what cannot.

The basic rule is that anything defined in one of the library's source files is *domestic*, and the rest is *foreign*. If for some reason, the source file is unknown, then, the definition will be considered domestic if the symbol naming it is from a domestic package.

This view on domesticity has implications on what is documented, which are worth mentioning.

- Language extensions (for example, new methods on standard generic functions) and, in general, new local definitions on symbols from foreign packages are considered domestic, and hence will appear in the reference manual.
- Conversely, extensions to the library, but defined outside of it, will *not* be part of the resulting reference manual, as they will be considered foreign.

Note that even though they are not normally meant to appear in the final reference manual, `Declt` may be led to create foreign definitions (sometimes a lot of them) as well as domestic ones. For example, if your library adds a method to `make-instance`, `Declt` *will* internally create a (foreign) definition for the `make-instance` generic function. Foreign definitions are never complete however (otherwise, we would end up with a complete documentation for the whole Lisp image): only the information relevant to the library is kept around.

As said before, it is possible to include those (partial) foreign definitions in the documentation by using the `:foreign-definitions` option to `declt` or `assemble`. Continuing with the previous example, local methods on `make-instance` normally appear as toplevel, standalone definitions in the documentation, as they belong to a foreign generic function. If, however, foreign definitions are included in the documentation, there will be a toplevel (partial) entry for `make-instance`, and the domestic methods will appear inside it (truly foreign methods will *never* be documented).

6.1.2 Introspection Levels

In order to discover both domestic and relevant foreign definitions, `Declt` uses a heuristic which establishes different tradeoffs between performance and accuracy. Remember that this is controlled by the `:introspection-level` option to `declt` or `assess`. This heuristic is based on the assumption that most libraries out there are implemented within packages of their own.

Consequently, this is what happens by default (that is, at introspection level 1). After the library has been loaded, `Declt` scans all live packages and selects the domestic ones. It then scans all symbols from these packages to find domestic definitions.

Note that if we were to stop there, we would probably miss many domestic definitions, namely, those on foreign symbols such as methods on standard generic functions such as `make-instance`. Fortunately, a lot of these will be recovered later on anyway, when `Declt` computes cross-references between definitions in the so-called *finalization* phase.

Suppose for instance that the library provides a new method on `make-instance`. This method is domestic because it is defined in one of the library's files, but at introspection level 1, `Declt` only scans domestic symbols, so it will miss it (`common-lisp`, the home package of the `make-instance` symbol, is foreign). On the other hand, it is very likely that this method exists because it specializes on a domestic class already known to `Declt`. When `Declt` finalizes a class definition, it looks up methods that specialize on it (among other things). These methods are in fact accessible through the class by introspection (they are called "direct methods"). Consequently, `Declt` *will* eventually discover the new `make-instance` method, figure out that it is a domestic one, and add it to the documentation.

In the end, it is probable that most domestic definitions end up being discovered, either during the initial scanning phase, or later on, during the finalization phase when cross-references are computed. This is why the default introspection level is probably good enough most of the time.

On the other hand, some (rare?) domestic definitions may still escape the discovery process at that level. For example, if you define a foreign global variable in one of your library's files, chances are that there won't be any domestic definition cross-referencing it. This is why `Declt` provides a second level of introspection, in which it initially scans *all* symbols in the Common Lisp image, rather than just the ones from domestic packages. The resulting documentation will then be more complete, although at the expense of a *much* greater computation time.

And yes, you have just noticed that I said "more complete" rather than "exhaustive", and mind you, this is not because my vocabulary is limited. The thing is that even with an initial scan of all symbols, we may still miss some information. To be precise, we won't miss any domestic definition. It is the cross-referencing information that may remain incomplete, and that is because the finalization phase doesn't re-scan each and every symbol in the Lisp image again; only the definitions that have been created so far. Let's take an example (granted, a contrived one; but aren't they all?). You define a regular domestic function, and it is used as the update function in a foreign setf expander. Even with an initial scan of all symbols in the Lisp image, you'll get a definition for your domestic function, but that's it. Later on, during the finalization process, there's no way to go from the function to the setf expander, so that information remains unknown. In order to fix that, the finalization process would need to re-scan the whole Lisp world again, and that would become introspection level 3. Not sure it's actually worth it; maybe one day...

6.1.3 Reports

As mentioned before, the `assess` function returns an instance of the `report` class. In case you want to manipulate reports directly, a complete description of this class, its slots and accessors is provided in the `Declt` *Reference Manual*. Several aspects of reports and their contents deserve a special mention however. This is the purpose of this section.

6.1.3.1 Definitions List

The most important slot in a `report` instance is the `definitions` one. This is the slot which contains the whole list of definitions (again, a *definition* is the abstract description of something that needs to be documented). Because reports do not make any assumption on the final organization of the reference manual (that is the job of the assembly stage of the `Declt` pipeline; see Section 6.2 [Assembly], page 21), the list of definitions is completely flat and mostly unorganized.

In terms of organization, the only guarantee you have is that the library's main system definition comes first, and that does not even mean first in the list (although it currently does), but first among all system definitions. Apart from that, you shouldn't count on any kind of ordering within that list.

Also, the list being flat means that every single definition appears as a toplevel element. In other words, it really is a list of definitions, as opposed to a tree of definitions. For instance, things that are normally nested, such as ASDF modules or files (being part of a super-component), slot definitions (being part of a class, structure, or condition), methods (being part of a generic function), *etc.*, are directly accessible in that list. You need to remember this if you ever need to filter out the definitions you're not interested in.

On the other hand, this list being mostly unorganized and flat doesn't mean that the programmatic order of things is completely lost. Many, if not all definitions, contain cross-references to other definitions. For example, ASDF system definitions point back to their dependencies, classes to their slots, generic functions to their methods, *etc.* In all these cases, the original definitions order is preserved.

6.1.3.2 Definitions

Definitions are implemented as an object-oriented hierarchy rooted in the `definition` class. The complete description of this hierarchy is provided in the `Declt` *Reference Manual*. This section provides an overview of the hierarchy in question, along with the most important remarks about it. Every definition class in the hierarchy has a name of the form `foo-definition`. For concision, and except for the `definition` and mixin classes, all class names in the figures below are abbreviated: a `foo-definition` class is simply denoted as `foo`. Abstract classes are drawn in red boxes with bold text, regular classes in blue boxes with normal font, and mixins (also abstract) in green ellipses with bold text.

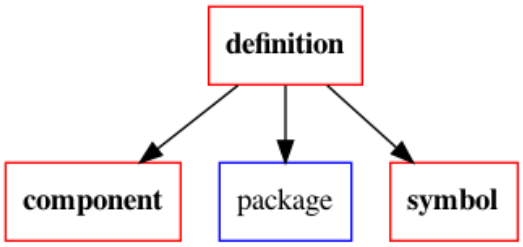


Figure 6.1: Main definition categories

As depicted in Figure 6.1, there are three main categories of definitions: `component-definition` is the root class for ASDF components, `package-definition` is the class for Lisp packages, and `symbol-definition` is the root class of all programmatic definitions named by symbols. The `definition` root class provides common properties, such as a reference to the corresponding Lisp object (when applicable), source information, and foreign status.

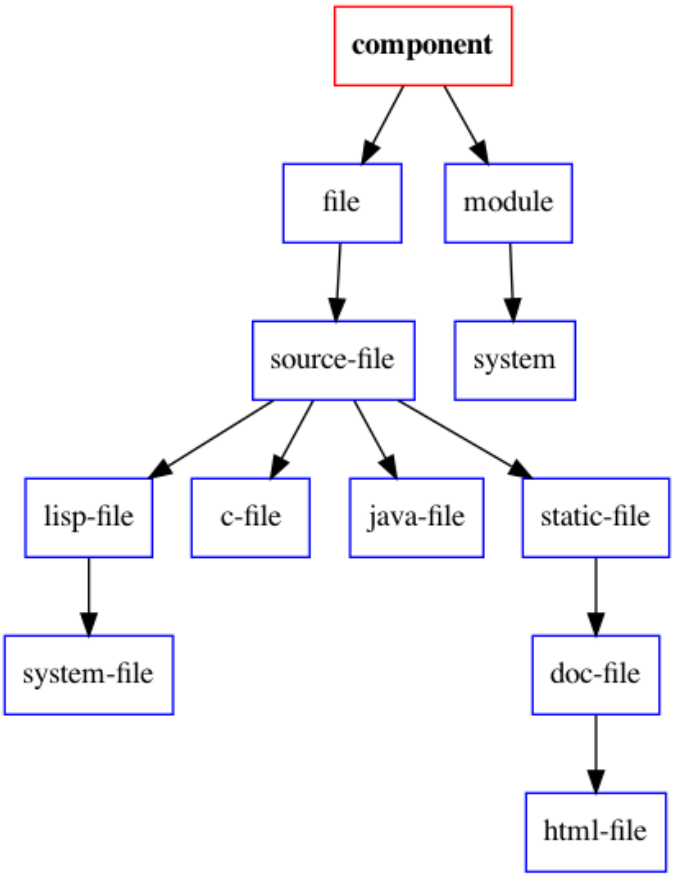


Figure 6.2: ASDF component definitions

Figure 6.2 depicts the hierarchy of ASDF definitions. Unsurprisingly, this hierarchy closely follows that of actual ASDF components, although some peculiarities are worth mentioning here.

- The `lisp-file-definition` class corresponds to the ASDF `cl-source-file` one. Note that ASDF provides two additional subclasses for Lisp files, with extensions '`.cl`' and '`.lsp`'. We don't do that, however. Instead, there is a protocol for retrieving a file component's extension.

- Notice the existence of a `system-file-definition` class, which is a subclass of `lisp-file-definition`. In ASDF, system files are *not* represented as components, but `Declt` pretends they are by faking a particular class of Lisp files for them. This allows us to document `.asd` files as particular Lisp files, without too much specific code. Note also that some (few) libraries list their system files (for example as static files) in the system definition. `Declt` is aware of this and removes such definitions, so as to avoid duplication.

The hierarchy of definitions named by symbols (under the `symbol-definition` class) is quite large, so it is split into four figures.

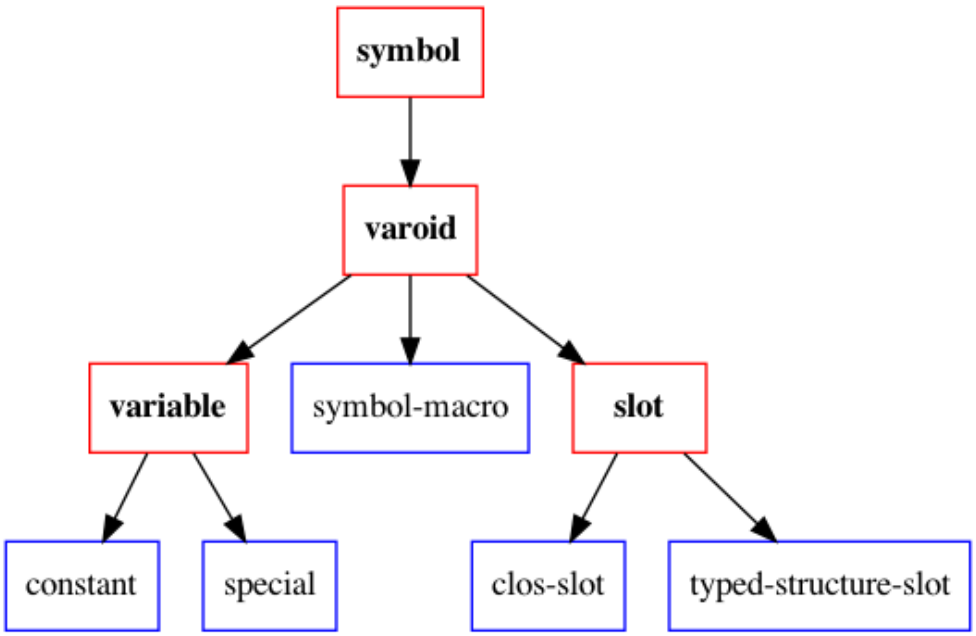


Figure 6.3: Varoid definitions

The *varoid* ones (Figure 6.3) represent simple values having symbolic names. This boils down to variables, symbol macros, and slots.

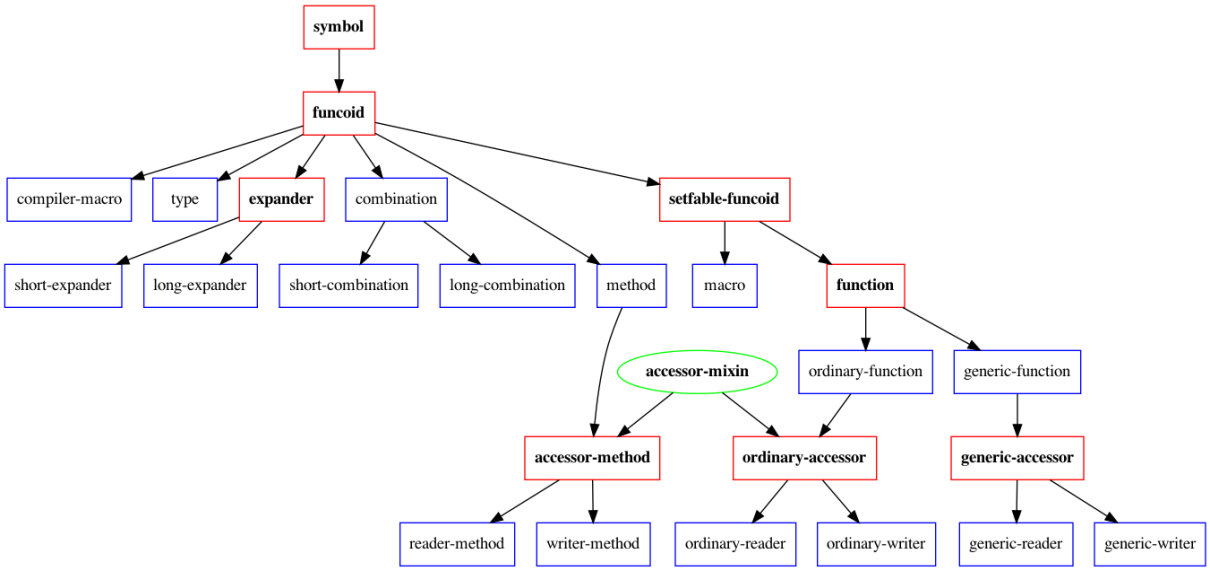


Figure 6.4: Funcoid definitions

The *funcoid* ones (Figure 6.4) covers "parametric values", that is, values having a lambda-list of some sort. This includes all variations on functions, macros, methods, types, setf expanders, and method combinations. You may wonder why the `combination-definition` class is not abstract. That is because the standard method combination is an instance of it. All funcoids have a slot indicating whether the definition is a setf one (*e.g.* a function named `(setf foo)`). The "setfable" branch represents definitions that may be related to a setf expander (that is, which could be either an *access-fn* or an *update-fn*). As a matter of fact, this hierarchy is not

100% correct: some funcoids can never be setf ones, and some setfable ones can never be related to a setf expander. On the other hand, doing it like this keeps said hierarchy relatively simple.

The "accessor" sub-branches represent definitions for methods, ordinary, or generic functions, which have been identified as reading or writing slots. The accessor mixin provides a back reference to the slot definition in question. Note that generic functions do not access slots; only their methods do (which is why the generic accessor branch does not use the accessor mixin). A generic function will be qualified as a *generic reader* (respectively a *generic writer*) if all its methods have been identified as reader (respectively writer) methods.

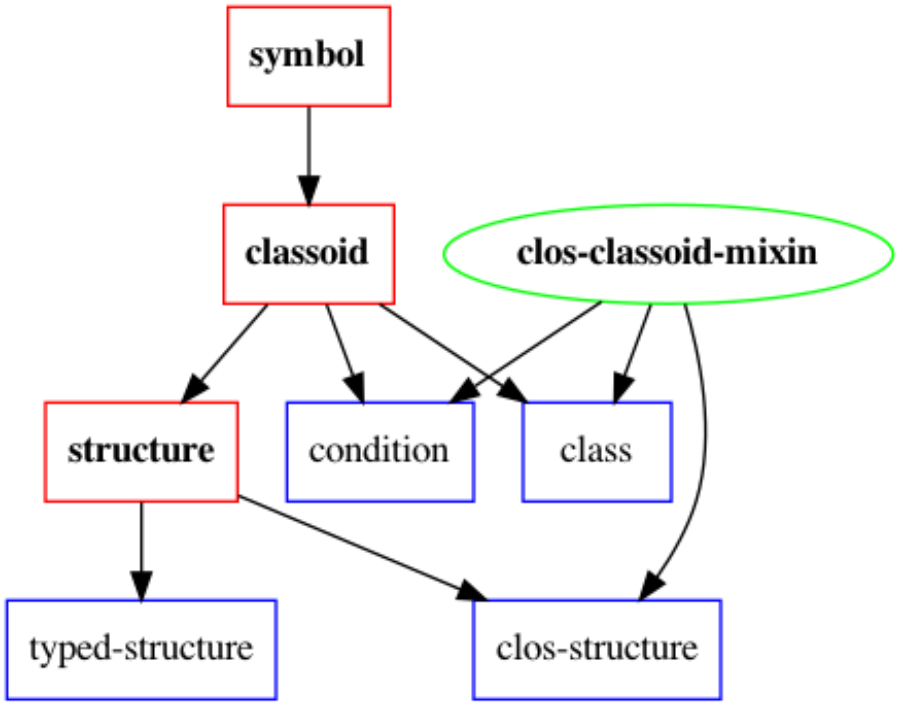


Figure 6.5: Classoid definitions

The *classoid* branch (Figure 6.5) represents (typed) structures, classes, and conditions. This part of the definitions hierarchy is probably not portable, as it relies on how SBCL implements a number of things, in particular, which classoids are in fact CLOS classes, which is represented by the CLOS classoid mixin. The CLOS classoid mixin provides super- and sub-classoid information, but this may change in the future. Indeed, it is perhaps possible to trace back `:include` information for typed structures as well, but this is not currently implemented.

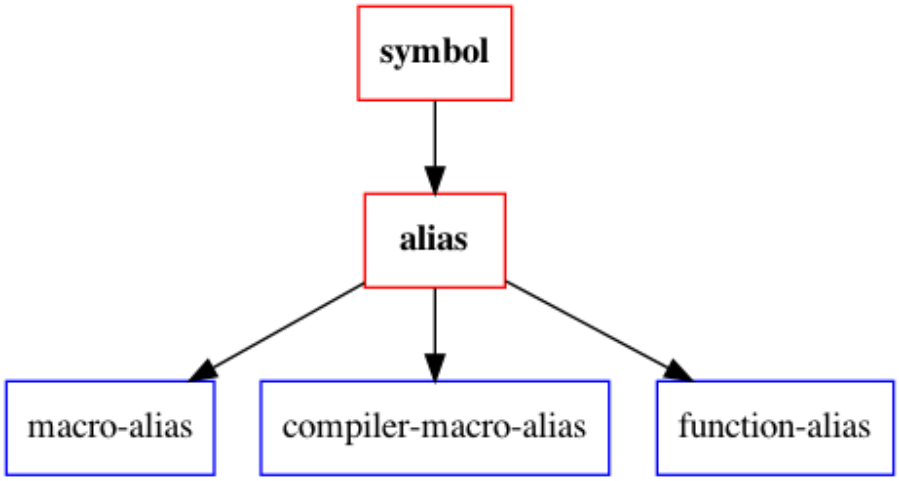


Figure 6.6: Alias definitions

Finally, the last group of symbol definitions is for *aliases*, that is, globally defined (compiler) macros or functions that are also manually attached to other symbols. Aliases are known to `Declt` so as to avoid duplicating documentation.

6.2 Assembly

6.3 Typesetting

7 Other Considerations

This section contains marginal or meta-information, orthogonal to the library's main purpose.

7.1 Configuration

Some aspects of `Declt`'s behavior can be configured *before* the library system is actually loaded. `Declt` stores its user-level configuration (along with some other setup parameters) in another ASDF system called '`net.didierverna.declt.setup`' (and the eponym package). In order to configure the library (I repeat, prior to loading it), you will typically do something like this:

```
(require "asdf")
(asdf:load-system :net.didierverna.declt.setup)
(net.didierverna.declt.setup:configure <option> <value>)
```

`configure` *KEY VALUE* [Function]
Set *KEY* to *VALUE* in the current `Declt` configuration.

Out of curiosity, you can also inquire the current configuration for specific options with the following function.

`configuration` *KEY* [Function]
Return *KEY*'s value in the current `Declt` configuration.

Currently, the following options are provided.

`:swank-eval-in-emacs`
: This option is only useful if you use Slime, and mostly if you plan on hacking `Declt` itself. The library provides indentation information for some of its functions directly embedded in the code. This information can be automatically transmitted to Emacs when the ASDF system is loaded if you set this option to `t`. However, note that for this to work, the Slime variable `slime-enable-evaluate-in-emacs` must also be set to `t` in your Emacs session. If you're interested to know how this process works, I have described it in this blog entry (`http://www.didierverna.net/blog/index.php?post/2011/07/20/One-more-indentation-hack`).

7.2 Version Numbering

As `Declt` evolves over time, you might one day feel the need for conditionalizing your code on the version of the library.

The first thing you can do to access the current version number of `Declt` is use the `version` function.

`version` **&optional** (*TYPE* **:number**) [Function]
Return the current version number of `Declt`. *TYPE* can be one of `:number`, `:short` or `:long`. For `:number`, the returned value is a fixnum. Otherwise, it is a string.

A `Declt` version is characterized by 4 elements as described below.

- A major version number stored in the parameter `*release-major-level*`.
- A minor version number, stored in the parameter `*release-minor-level*`.
- A release status stored in the parameter `*release-status*`. The status of a release can be `:alpha`, `:beta`, `:rc` (standing for "release candidate") or `:patchlevel`. These are in effect 4 levels of expected stability.
- A status-specific version number stored in the parameter `*release-status-level*`. Status levels start at 1 (alpha 1, beta 1 and release candidate 1) except for stable versions, in which case patch levels start at 0 (*e.g.* 2.4.0).

In addition to that, each version of `Declt` (in the sense *major.minor*, regardless of the status) has a name, stored in the parameter `*release-name*`. The general naming theme for `Declt` is "Star Trek Characters".

Here is how the `version` function computes its value.

- A version `:number` is computed as *major* `*` *10000* `+` *minor* `*` *100* `+` *patchlevel*, effectively leaving two digits for each level. Note that alpha, beta and release candidate status are ignored in version numbers (this is as if the corresponding status level was considered to be always 0). Only stable releases have their level taken into account.
- A `:short` version will appear like this for unstable releases: 1.3a4, 2.5b8 or 4.2rc1. Remember that alpha, beta or release candidate levels start at 1. Patchlevels for stable releases start at 0 but 0 is ignored in the output. So for instance, version 4.3.2 will appear as-is, while version 1.3.0 will appear as just 1.3.
- A `:long` version is expanded from the short one, and includes the release name. For instance, 1.3 alpha 4 `"Uhura"`, 2.5 beta 8 `"Scotty"`, 4.2 release candidate 1 `"Spock"` or 4.3.2 `"Counselor Troy"`. As for the short version, a patchlevel of 0 is ignored in the output: 1.3 `"Uhura"`.

Incidentally, but you will probably never need to use it, `Declt` also exports a variable named `*copyright-years*`, which, as its name suggests, is a string denoting the copyright years for the whole project.

8 Conclusion

So that's it I guess. You know all about `Declt` now. The next step is to polish your own libraries so that `Declt` can extract meaningful documentation from them.

Then, you will want to run `Declt` on all the other libraries you use, in order to finally know how they work.

Now, go my friend. Go document the whole Lisp world!

Appendix A Documentation Tuning

A.1 Coding Style

Some elements of your own coding style will affect the reference manuals generated by `Declt`. This section provides some recommendations that will make the generated output look nicer.

A.1.1 Taglines

Unless you provide it with an explicit `:tagline` argument, `declt` uses the system's long name or description (provided they are one-liners) to construct a subtitle. Consequently, it is advisable to use a single (short) line of text for these slots. The system's long name should typically be the expansion of the system's name, if that's an acronym, or be left to `nil`.

A.1.2 Docstrings

`Declt` tries to make the generated output look nicer in various ways. For example, setf functions are documented right after the corresponding reader (if any) instead of being listed under the "S" letter. In a similar vein, methods are documented as components of their respective generic function, not as toplevel definitions (except for foreign methods without a corresponding generic function definition).

One thing that you can influence is `Declt`'s attempt at merging definitions. Merging may occur when there are definitions for both `symbol` and `(setf symbol)`. This happens for accessor functions, generic functions or setf expanders. This also happens for accessor methods. If possible, `Declt` will try to generate a *single* definition for both the reader and the writer. That is only possible, however, if both definitions would render the same documentation, *i.e.* same package, source file and docstring.

If you don't provide a docstring, merging will work. If you provide different docstrings (like "Set the value of ..." and "Get the value of ..."), you will effectively prevent merging from happening. One thing I like to do is to provide the *same* neutral docstring for readers and writers. For instance "Access the value of ...". This way, definitions can both provide a docstring and be merged together.

A.2 Pretty Printing

All text coming from either Common Lisp or one of `declt`'s initialization arguments (`:introduction` and `:conclusion` excepted) is properly escaped for the Texinfo format, so you don't need to worry about that.

In non-verbatim contexts, `Declt` attempts to pretty-print symbols the names of which would otherwise be problematic. In particular, the empty symbol (`||`) is denoted as the math empty set, and blank characters (spaces, tabs, and newlines), are replaced by more explicit Unicode symbols. Note that this means that even for Info output, a Unicode reader is required.

`Declt` also attempts to do some pretty printing on things like docstrings, system long description *etc.*. Currently, there's a simple heuristic that tries to detect short lines that should probably stand on their own (with an explicit line break). For every piece of text, `Declt` first calculates the longest line and bases its line breaking decision on that. In the future, it will be possible to bypass this calculation by specifying an intended line width. Other pretty printers will likely be made available as well.

A.3 Caveats

`Declt` currently has two main limitations that you need to understand in order to avoid bad surprises, plus some less serious design decisions that are still worth knowing.

A.3.1 SBCL Only

First, `Declt` is an SBCL-only library. That is because it relies on `sb-introspect`. This limitation may be lifted in the future by using equivalent API from other Common Lisp implementations, but in the meantime, this means two things.

1. `Declt` can only document libraries that work with SBCL, because it needs to load them (see Chapter 1 [Introduction], page 3).
2. If your ASDF system contains vendor-specific modules or components, `Declt` will only be able to document SBCL-specific ones.

Note that more generally, `Declt` only documents modules or components that ASDF actually loads, so if your system definition contains some form of conditional inclusion, this will affect the generated documentation.

A.3.2 Method Combinations

The method combination interface in Common Lisp is underspecified. In particular, although you define method combinations globally, changing them afterwards may not affect already created generic functions. As a result, you could in theory end up with *many* different method combinations with the same name, used in various generic functions. See this blog entry (`http://www.didierverna.net/blog/index.php?post/2013/08/16/Lisp-Corner-Cases%3A-Method-Combinations`) and this article (`https://www.lrde.epita.fr/~didier/research/publications/papers.php#verna.18.els`) for more explanations.

`Declt` assumes however that you have some sanity and only define method combinations once per name. They are documented as top level items and generic functions using them provide cross-references.

A.3.3 Anchor Names

Because of the Texinfo anchor syntax, some characters are very problematic (or even completely prohibited) in anchor names. In order to provide robust anchors, `Declt` transforms those characters into Unicode alternatives, resembling the original ones, but not quite the same. This is normally not a problem, but you may get bitten by this if you ever happen to attempt a copy/paste operation from an Info anchor or an HTML hyperlink (PDF links are fine): while the link may look like the original Lisp symbol, some characters will actually be different. More specifically, dots, commas, colons, parentheses, and backslashes will be replaced by close Unicode symbols. Additionally, the ampersand is turned upside down, so this one will be noticeable. . .

Appendix B Supported Platforms

`Declt` requires ASDF 3 and SBCL 2.1.2 or later. Other Lisp implementations are not currently supported, but may be in the future. The Texinfo code that `Declt` generates requires Makeinfo 6.7, and contains extended Unicode characters (so even for reading Info, Unicode support is necessary).

In order to compile and install the user manuals on your own, you will need `makeinfo` / `install-info`, `convert` (from Image Magick), `dot`, and `graph-easy` (from the Graph::Easy Perl library).

Appendix C Indexes

C.1 Concepts

C.2 Functions

A

C

D

N

T

V

C.3 Variables

*

N

P

S

C.4 Data Types

Appendix D Acknowledgments

The following people have contributed bug reports or fixes, suggestions, compiler support or any other kind of help. You have my gratitude!

BR. Fenn Pocock
Sabra Crolleton
Robert Goldman
“Symbolics”
Gavin Smith