# PAX MANUAL

## Contents

[in package MGL-PAX with nicknames PAX]

# 1 Introduction

*What if documentation really lived in the code?*

Docstrings are already there. If some narrative glued them together, we'd be able develop and explore the code along with the documentation due to their physical proximity. The main tool that PAX provides for this is `defsection`:

```
(defsection @foo-random-manual (:title "Foo Random manual")
  "Foo Random is a random number generator library."
  (foo-random-state class)
  (uniform-random function)
  (@foo-random-examples section))
```

Like this one, sections can have docstrings and references to definitions (e.g. `(uniform-random function)`). These docstrings and references are the glue. To support interactive development, PAX

- makes SLIME's `M-.` work with references and
- adds a documentation browser.

See Emacs Setup.

Beyond interactive workflows, Generating Documentation from sections and all the referenced items in Markdown or HTML format is also implemented.

With the simplistic tools provided, one may emphasize the narrative as with Literate Programming, but documentation is generated from code, not vice versa, and there is no support for chunking.

*Code is first, code must look pretty, documentation is code*.

**Docstrings** PAX automatically recognizes and marks up code with backticks and links names in code to their definitions. Take, for instance, SBCL's `abort` function, whose docstring is written in the usual style, uppercasing names of symbols:

```
(docstring #'abort)
=> "Transfer control to a restart named ABORT, signalling
a CONTROL-ERROR if none exists."
```

Note how in the generated documentation, `abort` is set with a monospace font, while `control-error` is Autolinked:

- [function] **ABORT** *&OPTIONAL CONDITION*

  Transfer control to a restart named `abort`, signalling a `control-error` if none exists.

The following transcript shows the raw Markdown for the previous example.

```
(document #'abort :format :markdown)
.. - [function] **ABORT** *&OPTIONAL CONDITION*
..
..     Transfer control to a restart named `ABORT`, signalling
..     a [`CONTROL-ERROR`][7c2c] if none exists.
..
..   [7c2c]: http://www.lispworks.com/documentation/HyperSpec/Body/e_contro.htm
↪  "CONTROL-ERROR (MGL-PAX:CLHS CONDITION)"
..
```

**A Complete Example** Here is an example of how it all works together:

```
(mgl-pax:define-package :foo-random
  (:documentation "This package provides various utilities for random.
  See FOO-RANDOM:@FOO-RANDOM-MANUAL.")
  (:use #:common-lisp #:mgl-pax))

(in-package :foo-random)

(defsection @foo-random-manual (:title "Foo Random manual")
  "FOO-RANDOM is a random number generator library inspired by CL:RANDOM.
  Functions such as UNIFORM-RANDOM use *FOO-STATE* and have a
  :RANDOM-STATE keyword arg."
  (foo-random-state class)
  (state (reader foo-random-state))
  "Hey we can also print states!"
  (print-object (method () (foo-random-state t)))
  (*foo-state* variable)
  (gaussian-random function)
  (uniform-random function)
  ;; This is a subsection.
  (@foo-random-examples section))

(defclass foo-random-state ()
  ((state :reader state)))

(defmethod print-object ((object foo-random-state) stream)
  (print-unreadable-object (object stream :type t)))

(defvar *foo-state* (make-instance 'foo-random-state)
  "Much like *RANDOM-STATE* but uses the FOO algorithm.")

(defun uniform-random (limit &key (random-state *foo-state*))
  "Return a random number from the between 0 and LIMIT (exclusive)
  uniform distribution."
  nil)

(defun gaussian-random (stddev &key (random-state *foo-state*))
  "Return a random number from a zero mean normal distribution with
  STDDEV."
  nil)

(defsection @foo-random-examples (:title "Examples")
  "Let's see the transcript of a real session of someone working
  with FOO:

  ```cl-transcript
  (values (princ :hello) (list 1 2))
  .. HELLO
  => :HELLO
  => (1 2)

  (make-instance 'foo-random-state)
  ==> #<FOO-RANDOM-STATE >
```
```

```
  ```")
```

Note how `(variable *foo-state*)` in the `defsection` form both exports `*foo-state*` and includes its documentation in `@foo-random-manual`. The symbols `variable` and `function` are just two instances of locatives, which are used in `defsection` to refer to definitions tied to symbols.

`(document @foo-random-manual)` generates fancy Markdown or HTML output with automatic markup and Autolinks uppercase words found in docstrings, numbers sections, and creates a table of contents.

One can even generate documentation for different but related libraries at the same time with the output going to different files but with cross-page links being automatically added for symbols mentioned in docstrings. In fact, this is what PAX World does. See Generating Documentation for some convenience functions to cover the most common cases.

The transcript in the code block tagged with `cl-transcript` is automatically checked for up-to-dateness when documentation is generated.

## 2 Emacs Setup

Here is a quick recipe for setting up PAX for use via SLIME to take advantage of the conveniences on offer. Conversely, there is no need to do any of this just to use `defsection`, write docstrings and for Generating Documentation.

If PAX was installed from Quicklisp, then evaluate this in CL to copy the Elisp code to a stable location:

```
(mgl-pax:install-pax-elisp "~/quicklisp/")
```

Assuming the Elisp file is in the `~/quicklisp/` directory, add something like this to your `.emacs`:

```
(add-to-list 'load-path "~/quicklisp/")
(require 'mgl-pax)
(global-set-key (kbd "C-.") 'mgl-pax-document)
(global-set-key (kbd "s-x t") 'mgl-pax-transcribe-last-expression)
(global-set-key (kbd "s-x r") 'mgl-pax-retranscribe-region)
```

### 2.1 Functionality Provided

- For Navigating Sources in Emacs, loading `mgl-pax` extends `slime-edit-definitions` (`M-.`) by adding `mgl-pax-edit-definitions` to `slime-edit-definition-hooks`. There are no related variables to customize.
- For Browsing Live Documentation, `mgl-pax-browser-function` and `mgl-pax-web-server-port` can be customized in Elisp. To browse within Emacs, choose `w3m-browse-url` (see w3m), and make sure both the w3m binary and the w3m Emacs package are installed. On Debian, simply install the `w3m-el` package. With other browser functions, a HUNCHENTOOT web server is started.
```

- See Transcribing with Emacs for how to use the transcription features. There are no related variables to customize.

## 2.2 Installing from Quicklisp

If you installed PAX with Quicklisp, the location of `mgl-pax.el` may change with updates, and you may want to copy the current version of `mgl-pax.el` to a stable location by evaluating this in CL:

```
(mgl-pax:install-pax-elisp "~/quicklisp/")
```

If working from, say, a git checkout, there is no need for this step.

- **[function]** **install-pax-elisp** *target-dir*

  Copy `mgl-pax.el` distributed with this package to `target-dir`.

## 2.3 Loading PAX

Assuming the Elisp file is in the `~/quicklisp/` directory, add something like this to your `.emacs`:

```
(add-to-list 'load-path "~/quicklisp/")
(require 'mgl-pax)
```

If the Lisp variable `mgl-pax-autoload` is true (the default), then MGL-PAX will be loaded in the connected Lisp on-demand via SLIME.

If loading fails, `mgl-pax` will be unloaded from Emacs and any overridden Slime key bindings restored.

## 2.4 Setting up Keys

The recommended key bindings are this:

```
(global-set-key (kbd "C-.") 'mgl-pax-document)
(global-set-key (kbd "s-x t") 'mgl-pax-transcribe-last-expression)
(global-set-key (kbd "s-x r") 'mgl-pax-retranscribe-region)
```

The global key bindings above are global because their commands work in any mode. If that's not desired, one may bind `C-.` locally in all Slime related modes like this:

```
(slime-bind-keys slime-parent-map nil '(("C-." mgl-pax-document)))
```

If the customizable variable `mgl-pax-hijack-slime-doc-keys` is true, then upon loading `mgl-pax`, the following changes are made to `slime-doc-map` (assuming it's bound to `C-c C-d`):

- `C-c C-d a`: replaces `slime-apropos` with `mgl-pax-apropos`
- `C-c C-d z`: replaces `slime-apropos-all` with `mgl-pax-apropos-all`
- `C-c C-d p`: replaces `slime-apropos-package` with `mgl-pax-apropos-package`
- `C-c C-d d`: replaces `slime-describe-symbol` with `mgl-pax-document`

- `C-c C-d f`: replaces `slime-describe-function` with `mgl-pax-document`
- `C-c C-d c`: installs `mgl-pax-current-definition-toggle-view`
- `C-c C-d u`: installs `mgl-pax-edit-parent-section`

Calling `mgl-pax-unhijack-slime-doc-keys` reverts these changes.

# 3 Links and Systems

Here is the official repository and the HTML documentation for the latest version.

PAX is built on top of the DRef library (bundled in the same repository).

- *Installation for deployment*

  The base system is mgl-pax. It has very few dependencies and is sufficient as a dependency for systems using the Basics to add documentation. This is to keep deployed code small. To install only the bare minimum, with no intention of using Navigating Sources in Emacs, Generating Documentation, Browsing Live Documentation or using Transcripts, under Quicklisp for example, PAX could be installed as:

  ```
  (ql:quickload "mgl-pax")
  ```

- *Installation for development*

  The heavier dependencies are on the other systems, which correspond to the main functionalities provided, intended to be used primarily during development. To install the dependencies for all features under Quicklisp, do

  ```
  (ql:quickload "mgl-pax/full")
  ```

  Having thus installed the dependencies, it is enough to load the base system, which will autoload the other systems as necessary.

- **[system]** **"mgl-pax"**
  - *Version:* 0.4.1
  - *Description:* Documentation system, browser, generator. See the PAX Manual.
  - *Long Description:* The base system. See Links and Systems.
  - *Licence:* MIT, see COPYING.
  - *Author:* Gábor Melis
  - *Mailto:* mega@retes.hu
  - *Homepage:* http://github.com/melisgl/mgl-pax
  - *Bug tracker:* https://github.com/melisgl/mgl-pax/issues
  - *Source control:* GIT
  - *Depends on:* dref, mgl-pax-bootstrap, named-readtables, pythonic-string-reader

    - *Defsystem depends on:* mgl-pax.asdf
- **[system]** **"mgl-pax/navigate"**
    - *Description:* Support for Navigating Sources in Emacs via Slime's `M-.` in MGL-PAX.
    - *Depends on:* alexandria, dref/full, mgl-pax, swank(?)
    - *Defsystem depends on:* mgl-pax.asdf
- **[system]** **"mgl-pax/document"**
    - *Description:* Support for Generating Documentation in MGL-PAX.
    - *Depends on:* 3bmd, 3bmd-ext-code-blocks, alexandria, colorize, md5, mgl-pax/navigate, mgl-pax/transcribe, trivial-utf-8
    - *Defsystem depends on:* mgl-pax.asdf
- **[system]** **"mgl-pax/web"**
    - *Description:* Web server for Browsing Live Documentation in MGL-PAX.
    - *Depends on:* hunchentoot, mgl-pax/document
    - *Defsystem depends on:* mgl-pax.asdf
- **[system]** **"mgl-pax/transcribe"**
    - *Description:* Support for Transcripts in MGL-PAX.
    - *Depends on:* alexandria, mgl-pax/navigate
    - *Defsystem depends on:* mgl-pax.asdf
- **[system]** **"mgl-pax/full"**
    - *Description:* The `mgl-pax` system with all features preloaded.
    - *Depends on:* mgl-pax/document, mgl-pax/navigate, mgl-pax/transcribe, mgl-pax/web

# 4 Background

As a user, I frequently run into documentation that's incomplete and out of date, so I tend to stay in the editor and explore the code by jumping around with SLIME's `M-.` (`slime-edit-definition`). As a library author, I spend a great deal of time polishing code but precious little writing documentation.

In fact, I rarely write anything more comprehensive than docstrings for exported stuff. Writing docstrings feels easier than writing a separate user manual, and they are always close at hand during development. The drawback of this style is that users of the library have to piece the big picture together themselves.

That's easy to solve, I thought, let's just put all the narrative that holds docstrings together in the code and be a bit like a Literate Programmer turned inside out. The original prototype, which did almost everything I wanted, was this:

```
(defmacro defsection (name docstring)
  `(defun ,name () ,docstring))
```

Armed with this `defsection`, I soon found myself organizing code following the flow of user-level documentation and relegated comments to implementation details entirely. However, some parts of `defsection` docstrings were just listings of all the functions, macros and variables related to the narrative, and this list was repeated in the `defpackage` form complete with little comments that were like section names. A clear violation of OAOO, one of them had to go, so `defsection` got a list of symbols to export.

That was great, but soon I found that the listing of symbols is ambiguous if, for example, a function, a compiler macro and a class were named by the same symbol. This did not concern exporting, of course, but it didn't help readability. Distractingly, on such symbols, `M-.` was popping up selection dialogs. There were two birds to kill, and the symbol got accompanied by a type, which was later generalized into the concept of locatives:

```
(defsection @introduction ()
  "A single line for one man ..."
  (foo class)
  (bar function))
```

After a bit of elisp hacking, `M-.` was smart enough to disambiguate based on the locative found in the vicinity of the symbol, and everything was good for a while.

Then, I realized that sections could refer to other sections if there were a `section` locative. Going down that path, I soon began to feel the urge to generate pretty documentation as all the necessary information was available in the `defsection` forms. The design constraint imposed on documentation generation was that following the typical style of upcasing symbols in docstrings, there should be no need to explicitly mark up links: if `M-.` works, then the documentation generator shall also be able figure out what's being referred to.

I settled on Markdown as a reasonably non-intrusive format, and a few thousand lines later PAX was born. Since then, locatives and references were factored out into the DRef library to let PAX focus on `M-.` and documentation.

# 5 Basics

Now let's examine the most important pieces.

- **[macro]** **defsection** *name (&key (package '*package*) (readtable '*readtable*) (export t) title link-title-to (discard-documentation-p *discard-documentation-p*)) &body entries*

  Define a documentation section and maybe export referenced symbols. A bit behind the scenes, a global variable with `name` is defined and is bound to a `section` object. By convention, section names start with the character `@`. See Introduction for an example.

  **Entries**

  `entries` consists of docstrings and references in any order. Docstrings are arbitrary strings in markdown format.

References are xrefs given in the form (name locative). For example, (foo function) refers to the function foo, (@bar section) says that @bar is a subsection of this one. (baz (method () (t t t))) refers to the default method of the three argument generic function baz. (foo function) is equivalent to (foo (function)). See the DRef Introduction for more.

The same name may occur in multiple references, typically with different locatives, but this is not required.

The references are not located until documentation is generated, so they may refer to things yet to be defined.

**Exporting**

If export is true (the default), name and the names of references among entries which are symbols are candidates for exporting. A candidate symbol is exported if

- it is accessible in package, and
- there is a reference to it in the section being defined which is approved by exportable-reference-p.

See define-package if you use the export feature. The idea with confounding documentation and exporting is to force documentation of all exported symbols.

**Misc**

title is a string containing markdown or nil. If non-nil, it determines the text of the heading in the generated output. link-title-to is a reference given as an (name locative) pair or nil, to which the heading will link when generating HTML. If not specified, the heading will link to its own anchor.

When discard-documentation-p (defaults to *discard-documentation-p*) is true, entries will not be recorded to save memory.

- **[variable]** ***discard-documentation-p*** *nil*

  The default value of defsection's discard-documentation-p argument. One may want to set *discard-documentation-p* to true before building a binary application.

- **[macro]** **define-package** *package &rest options*

  This is like cl:defpackage but silences warnings and errors signalled when the redefined package is at variance with the current state of the package. Typically this situation occurs when symbols are exported by calling export (as is the case with defsection) as opposed to adding :export forms to the defpackage form and the package definition is subsequently reevaluated. See the section on package variance in the SBCL manual.

  The bottom line is that if you rely on defsection to do the exporting, then you'd better use define-package.

- **[macro]** **define-glossary-term** *name (&key title url (discard-documentation-p *discard-documentation-p*)) &body docstring*

Define a global variable with `name`, and set it to a `glossary-term` object. `title`, `url` and `docstring` are markdown strings or `nil`. Glossary terms are `document`ed in the lightweight bullet + locative + name/title style. See the glossary entry name for an example.

When a glossary term is linked to in documentation, its `title` will be the link text instead of the name of the symbol (as with `section`s).

Glossary entries with a non-`nil` `url` are like external links: they are linked to their `url` in the generated documentation. These offer a more reliable alternative to using markdown reference links and are usually not included in `section`s.

When `discard-documentation-p` (defaults to `*discard-documentation-p*`) is true, `docstring` will not be recorded to save memory.

- **[macro]** **note** *&body args*

  Define a note with an optional `name` and an optional `docstring`. The `docstring` of the note is its own docstring concatenated with docstrings of other notes in the lexical scope of `body`.

  `args` has the form [`name`] [`docstring`] `body`, where the square brackets indicate optional arguments. See below for the details of parsing `args`.

  **`note` is experimental and as such subject to change.**

  `note` can be wrapped around any expression that's evaluated without changing its run-time behaviour or introducing any run-time overhead. The names of notes live in the same global namespace regardless of nesting or whether they are top level forms. *These properties come at the price of `note` being weird: it defines named notes at macro-expansion time (or load time). But the definitions are idempotent, so it's fine to macroexpand `note` any number of times.*

  Notes are similar to Lisp comments, but they can be included in the documentation with the `docstring` locative. Notes are intended to help reduce the distance between code and its documentation when there is no convenient definition docstring to use nearby.

```
(note @xxx "We change the package."
  (in-package :mgl-pax))
==> #<PACKAGE "MGL-PAX">
(values (docstring (dref '@xxx 'note)))
=> "We change the package."
```

  Here is an example of how to overdo things:

```
(note @1+*
  "This is a seriously overdone example."
  (defun 1+* (x)
    "[@1+* note][docstring]"
    (if (stringp x)
        (note (@1+*/1 :join #\Newline)
          "- If X is a STRING, then it is parsed as a REAL number."
          (let ((obj (read-from-string x)))
            (note "It is an error if X does not contain a REAL."
              (unless (realp obj)
```

```
                (assert nil)))
            (1+ obj)))
        (note "- Else, X is assumed to be REAL number, and we simply
                add 1 to it."
          (1+ x)))))

(1+* "7")
=> 8

(values (docstring (dref '@1+* 'note)))
=> "This is a seriously overdone example.

- If X is a STRING, then it is parsed as a REAL number.
It is an error if X does not contain a REAL.

- Else, X is assumed to be REAL number, and we simply
add 1 to it."
```

The parsing of `args`:

- If the first element of `args` is not a string, then it is a `name` (a non-`nil` `symbol`) or name with options, currently destructured as `(name &key join)`. As in `defsection` and `define-glossary-term`, the convention is that `name` starts with a `@` character.

  `join` is `princ`ed before the docstring of a child note is output. Its default value is a string of two newline characters.

- The next element of `args` is a Markdown docstring. See Markdown in Docstrings.

- The rest of `args` is the `body`. If `body` is empty, then `nil` is returned.

Note that named and nameless notes can contain other named or nameless notes without restriction, but nameless notes without a lexically enclosing named note are just an implicit progn with `body`, and their docstring is discarded.

If `note` occurs as a top level form, then its `source-location` is reliably recorded. Else, the quality of the source location varies, but it is at least within the right top level form on all implementations. On SBCL, exact source location is supported.

# 6 Parsing

When encountering a word such as `CLASSes`, PAX needs to find the name in it that makes sense in the context. Codification, for example, looks for interesting names, Navigating Sources in Emacs for names with Lisp `definitions`, and Linking for names with any kind of definition.

This is not as straightforward as it sounds because it needs to handle cases like `nonREADable`, `classes`, all the various forms of Linking in docstrings as well as in comments, and the `(name locative)` syntax in `defsection`.

- **[glossary-term]** **word**

A *word* is a string from which we want to extract a name. When Navigating, the word is `slime-sexp-at-point` or the label of a Markdown reference link if point is over one. Similarly, when Generating Documentation, it is a non-empty string between whitespace characters in a docstring or the label of a Markdown reference link.

- **[glossary-term] raw name**

  A *raw name* is a string from which a name may be read. Raw names correspond to an intermediate parsing step between words an names. See Names in Raw Names.

- **[glossary-term] name**

  A *name* is a DRef name. That is, a symbol or a string associated with a definition, whose kind is given by a locative.

Depending on the context, trimming and depluralization may be enabled (see Raw Names in Words), while the possible names may be restricted to symbols (see Names in Raw Names).

- *Trimming:* Enabled for Navigating Sources in Emacs and Codification.
- *Depluralization:* Enabled when the word is part of the normal flow of text (i.e. not for Specific Reflink with Text, Unspecific Reflink with Text and various Elisp functions such as `mgl-pax-apropos` unless they determine their argument from buffer contents).
- *Symbols only:* This is the case for Codification and Unspecific Autolink to prevent string-based definitions from littering the documentation with links without the control provided by explicitly `importing` symbols.

For a word, a number of raw names is generated by trimming delimiter characters and plural markers, and for each raw name a number of names are considered until one is found suitable in the context. The following subsections describe the details of the parsing algorithm.

## 6.1 Raw Names in Words

From words, raw names are parsed by trimming some prefixes and suffixes. For a given word, multiple raw names are considered in the following order.

1. The entire word.
2. Trimming the following characters from the left of the word:

   ```
   #<{;"'`
   ```

3. Trimming the following characters from the right of the word:

   ```
   ,;:.>}"'`
   ```

4. Trimming both of the previous two at the same time.
5. From the result of 4., If a word ends with what looks like a plural marker (case-insensitive), then a name is created by removing it. For example, from the word `buses` the plural marker `es` is removed to produce the name `bus`. The list of plural markers considered is `ses` (e.g. `gasses`), `es` (e.g. `buses`), `s` (e.g. `cars`), `zes` (e.g. `fezzes`), and `ren` (e.g. `children`).

6. From the result of 4., removing the prefix before the first, and the suffix after the last uppercase character if it contains at least one lowercase character.

## 6.2 Names in Raw Names

For each raw name from Raw Names in Words, various names may be considered until one is found suitable in the context.

The following examples list the names considered for a given raw name, assuming that `readtable-case` is `:upcase` as well as that `foo` and `|Foo|` are interned.

- `"foo"`: `foo`, `"foo"`, `"FOO"` (rules 1, 2, 3)
- `"FOO"`: `foo`, `"FOO"` (rules 1, 2)
- `"Foo"`: `"Foo"`, `"FOO"` (rules 2, 3)
- `"|Foo|"`: `|Foo|` (rule 4)
- `"\"foo\""`: `"foo"` (rule 5)

The rules are:

1. If the raw name is not mixed case (i.e. it doesn't have both upper- and lowercase characters) and it names an interned symbol (subject to the current Package and Readtable), then that symbol is considered as a name.
2. The raw name itself (a string) is considered a name.
3. The raw name upcased or downcased according to `readtable-case` (subject to the current readtable) but still as a string. This is to allow `[dref][package]` to refer to the `"DREF"` package regardless of whether the symbol `dref` is interned in the current package.
4. If the raw name is explicitly a symbol (it starts with `#\|`), and it names an interned symbol (subject to the current Package and Readtable), then that symbol is considered as a name and nothing else.
5. If the raw name has an embedded string (it starts with `#\"`) and `read-from-string` can read the embedded string from it, then that string is considered as a name and nothing else.

For example, when `M-.` is pressed while point is over `nonREADable.`, the last word of the sentence `It may be nonREADable.`, the following raw names are considered until one is found with a definition:

1. The entire word, `"nonREADable."`.
2. Trimming left does not produce a new raw name.
3. Trimming right removes the dot and gives `"nonREADable"`.
4. Trimming both is the same as trimming right.
5. No plural markers are found.

6. The lowercase prefix and suffix is removed around the uppercase core, giving `"READ"`. This names an interned symbol which has a definition, so `M-.` will visit it.

When Generating Documentation, Autolinking behaves similarly.

## 7 PAX Locatives

To the Basic Locative Types defined by DRef, PAX adds a few of its own.

- **[locative]** **section**
    - Direct locative supertypes: variable

  Refers to a `section` defined by `defsection`.

  `section` is `exportable-locative-type-p` but not exported by default (see `exportable-reference-p`).

- **[locative]** **glossary-term**
    - Direct locative supertypes: variable

  Refers to a `glossary-term` defined by `define-glossary-term`.

  `glossary-term` is `exportable-locative-type-p` but not exported by default (see `exportable-reference-p`).

- **[locative]** **note**

  Refers to named notes defined by the `note` macro.

  If a single link would be made to a `note` (be it either a Specific Link or an unambiguous Unspecific Link), then the `note`'s `docstring` is included as if with the `docstring` locative.

  `note` is `exportable-locative-type-p` but not exported by default (see `exportable-reference-p`).

- **[locative]** **dislocated**

  Refers to a symbol in a non-specific context. Useful for suppressing Unspecific Autolinking. For example, if there is a function called `foo` then

  ```
  `FOO`
  ```

  will be linked (if `*document-link-code*`) to its definition. However,

  ```
  [`FOO`][dislocated]
  ```

  will not be. With a dislocated locative, `locate` always fails with a `locate-error` condition. Also see Escaping Autolinking.

  `dislocated` references do not `resolve`.

- **[locative]** **argument**

An alias for `dislocated`, so that one can refer to an argument of a macro without accidentally linking to a class that has the same name as that argument. In the following example, `format` may link to `cl:format` (if we generated documentation for it):

```
"See FORMAT in DOCUMENT."
```

Since `argument` is a locative, we can prevent that linking by writing:

```
"See the FORMAT argument of DOCUMENT."
```

`argument` references do not `resolve`.

- **[locative]** **include** *source &key line-prefix header footer header-nl footer-nl*

  This `pseudo` locative refers to a region of a file. `source` can be a `string` or a `pathname`, in which case the whole file is being pointed to, or it can explicitly supply `start`, `end` locatives. `include` is typically used to include non-lisp files in the documentation (say markdown or Elisp as in the next example) or regions of Lisp source files. This can reduce clutter and duplication.

```
(defsection @example-section ()
  (mgl-pax.el (include #.(asdf:system-relative-pathname
                          :mgl-pax "src/mgl-pax.el")
                       :header-nl "```elisp" :footer-nl "```"))
  (foo-example (include (:start (dref-ext:make-source-location function)
                         :end (dref-ext:source-location-p function))
                        :header-nl "```"
                        :footer-nl "```")))
```

  In the above example, when documentation is generated, the entire `src/mgl-pax.el` file is included in the markdown output surrounded by the strings given as `header-nl` and `footer-nl`. The documentation of `foo-example` will be the region of the file from the `source-location` of the `start` reference (inclusive) to the `source-location` of the `end` reference (exclusive). If only one of `start` and `end` is specified, then they default to the beginning and end of the file, respectively.

  Since `start` and `end` are literal references, pressing `M-.` on `pax.el` will open the `src/mgl-pax.el` file and put the cursor on its first character. `M-.` on `foo-example` will go to the source location of the `foo` function.

  With the `lambda` locative, one can specify positions in arbitrary files.

  - `source` is either an absolute pathname designator or a list matching the destructuring lambda list `(&key start end)`, where `start` and `end` must be `nil` or `(<name> <locative>)` lists (not evaluated) like a `defsection` entry. Their `source-location`s constitute the bounds of the region of the file to be included. Note that the file of the source location of `start` and `end` must be the same. If `source` is a pathname designator, then it must be absolute so that the locative is context independent.
  - If specified, `line-prefix` is a string that's prepended to each line included in the documentation. For example, a string of four spaces makes markdown think it's a code block.

- `header` and `footer`, if non-`nil`, are printed before the included string.
- `header-nl` and `footer-nl`, if non-`nil`, are printed between two `fresh-line` calls.

`include` is not `exportable-locative-type-p`, and `include` references do not `resolve`.

- **[locative] docstring**

  `docstring` is a `pseudo` locative for including the parse tree of the markdown `docstring` of a definition in the parse tree of a docstring when generating documentation. It has no source location information and only works as an explicit link. This construct is intended to allow docstrings to live closer to their implementation, which typically involves a non-exported definition.

```
(defun div2 (x)
  "X must be [even* type][docstring]."
  (/ x 2))

(deftype even* ()
  "an even integer"
  '(satisfies evenp))

(document #'div2)
.. - [function] DIV2 X
..
..     X must be an even integer.
..
```

  There is no way to `locate` `docstring`s, so nothing to `resolve` either.

- **[locative] go** *(name locative)*

  Redirect to a definition in the context of the reference designated by `name` and `locative`. This `pseudo` locative is intended for things that have no explicit global definition.

  As an example, consider this part of a hypothetical documentation of CLOS:

```
  (defsection @clos ()
    (defmethod macro)
    (call-next-method (go (defmethod macro))))
```

  The `go` reference exports the symbol `call-next-method` and also produces a terse redirection message in the documentation.

  `go` behaves as described below.

  - A `go` reference `resolve`s to what `name` with `locative` resolves to:

```
(resolve (dref 'xxx '(go (print function))))
==> #<FUNCTION PRINT>
=> T
```

  - The `docstring` of a `go` reference is `nil`.
  - `source-location` (thus `M-.`) returns the source location of the embedded reference:

```
(equal (source-location (dref 'xxx '(go (print function))))
       (source-location (dref 'print 'function)))
=> T
```

- **[locative]** **clhs** *&optional nested-locative*

  Refers to definitions, glossary entries, sections, issues and issue summaries in the Common Lisp HyperSpec. These have no source location so `M-.` will not work. What works is linking in documentation, including Browsing Live Documentation. The generated links are relative to `*document-hyperspec-root*` and work even if `*document-link-to-hyperspec*` is `nil`. All matching is case-insensitive.

  - *definitions*: These are typically unnecessary as `document` will produce the same link for e.g. `PPRINT`, `[PPRINT][function]`, or `[pprint][]` if `*document-link-to-hyperspec*` is non-`nil` and the `pprint` function in the running Lisp is not linkable. When Browsing Live Documentation, a slight difference is that everything is linkable, so using the `clhs` link bypasses the page with the definition in the running Lisp.

    - *unambiguous definition*: `[pprint][clhs]` (pprint)
    - *disambiguation page*: `[function][clhs]` (function)
    - *specific*: `[function][(clhs class)]` (function)

  - *glossary terms*:

    - `[lambda list][(clhs glossary-term)]` (lambda list)

  - *issues*:

    - `[ISSUE:AREF-1D][clhs]` (ISSUE:AREF-1D)
    - `[ISSUE:AREF-1D][(clhs section)]` (ISSUE:AREF-1D)

  - *issue summaries*: These render as (SUMMARY:CHARACTER-PROPOSAL:2-6-5):

    - `[SUMMARY:CHARACTER-PROPOSAL:2-6-5][clhs]`
    - `[SUMMARY:CHARACTER-PROPOSAL:2-6-5][(clhs section)]`

    Since these summary ids are not particularly reader friendly, the anchor text a Specific Reflink with Text may be used:

    - `[see this][SUMMARY:CHARACTER-PROPOSAL:2-6-5 (clhs section)]` (see this).

  - *sections*:

    - *by section number*: `[3.4][clhs]` or `[3.4][(clhs section)]` (3.4)
    - *by section title* (substring match): `[lambda lists][clhs]` or `[lambda lists][(clhs section)]` (lambda lists)
    - *by filename*: `[03_d][clhs]` or `[03_d][(clhs section)]` (03_d)

    - *by alias*: Format directives are aliases of the sections describing them. Thus, `[~c][clhs]` is equivalent to `[22.3.1.1][clhs]` and `[Tilde C: Character][clhs]`. The full list is ~C ~% ~& ~| ~~ ~R ~D ~B ~O ~X ~F ~E ~G ~$ ~A ~S ~W ~_ ~< ~:> ~I ~/ ~T ~< Justification ~> ~* ~[ ~] ~{ ~} ~? ~( ~) ~P ~; ~^ ~Newline.

      Similarly, reader macro characters are aliases of the sections describing them. The full list is ( ) ' ; " ` , # #\ #' #( #* #: #. #B #O #X #R #C #A #S #P #= ## #+ #- #| #< #).

      Finally, loop keywords have aliases to the sections describing them. For example, the strings `loop:for`, `for` and `:for` are aliases of `clhs 6.1.2.1`. The `loop:*` aliases are convenient for completion at the prompt when Browsing Live Documentation, while the other aliases are for defaulting to buffer contents.

  As the above examples show, the `nested-locative` argument of the `clhs` locative may be omitted. In that case, definitions, glossary terms, issues, issue summaries, and sections are considered in that order. Sections are considered last because a substring of a section title can be matched by chance easily.

  All examples so far used Reflinks. Autolinking also works if the name is marked up as code or is codified (e.g. in `COS clhs` (`cos` clhs).

  As mentioned above, `M-.` does not do anything over `clhs` references. Slightly more usefully, the live documentation browser understands `clhs` links so one can enter inputs like `3.4 clhs`, `"lambda list" clhs` or `error (clhs function)`.

  `clhs` references do not `resolve`.

# 8 Navigating Sources in Emacs

Integration into SLIME's `M-.` (`slime-edit-definition`) allows one to visit the `source-location` of a definition. PAX extends standard Slime functionality by

- adding support for all kinds of definitions (see e.g. `asdf:system`, `readtable` in Basic Locative Types), not just the ones Slime knows about,
- providing a portable way to refer to even standard definitions,
- disambiguating the definition based on buffer content, and
- adding more powerful completions.

The definition is either determined from the buffer content at point or is prompted for. At the prompt, TAB-completion is available for both names and locatives. With a prefix argument (`C-u M-.`), the buffer contents are not consulted, and `M-.` always prompts.

The `M-.` extensions can be enabled by loading `src/mgl-pax.el`. See Emacs Setup. In addition, the Elisp command `mgl-pax-edit-parent-section` visits the source location of the section containing the definition with point in it.

A close relative of `M-.` is `C-.` for Browsing Live Documentation.

## 8.1 M-. Defaulting

When M-. is invoked, it first tries to find a name in the current buffer at point. If no name is found, then it prompts.

First, (slime-sexp-at-point) is taken as a word, from which the name will be parsed. Then, candidate locatives are looked for before and after the word. Thus, if a locative is the previous or the next expression, then M-. will go straight to the definition which corresponds to the locative. If that fails, M-. will try to find the definitions in the normal way, which may involve popping up an xref buffer and letting the user interactively select one of possible definitions.

M-. works on parenthesized references, such as those in defsection:

```
(defsection @foo ()
  (cos function))
```

Here, when the cursor is on one of the characters of cos or just after cos, pressing M-. will visit the definition of the function cos.

To play nice with Generating Documentation, forms suitable for Autolinking are recognized:

```
function cos
cos function
```

... as well as Reflinks:

```
[cos][function]
[see this][cos function]
```

... and Markdown inline code:

```
cos `function`
`cos` function
`cos` `function`
```

Everything works the same way in comments and docstrings as in code. In the next example, pressing M-. on resolve* will visit its denoted method:

```
;;; See RESOLVE* (method () (function-dref)) for how this all works.
```

## 8.2 M-. Prompting

### 8.2.1 M-. Minibuffer Syntax

At the minibuffer prompt, the definitions to edit can be specified as follows.

- name: Refers to all dref:definitions of name with a Lisp locative type. See these name -> definitions examples:

```
print    ->  PRINT FUNCTION
PRINT    ->  PRINT FUNCTION
MGL-PAX  ->  "mgl-pax" ASDF:SYSTEM, "MGL-PAX" package
pax      ->  "PAX" PACKAGE
"PAX"    ->  "PAX" PACKAGE
```

Note that depending on the Lisp implementation there may be more definitions. For example, SBCL has an `unknown :defoptimizer` definition for `print`.

- `name locative`: Refers to a single definition (as in `(dref:dref name locative)`). Example inputs of this form:

```
print function
dref-ext:docstring* (method nil (t))
```

- `locative name`: This has the same form as the previous: two sexps, but here the first one is the locative. If ambiguous, this is considered in addition to the previous one. Example inputs:

```
function print
(method nil (t)) dref-ext:docstring*
```

In all of the above `name` is a raw name, meaning that `print` will be recognized as `print` and `pax` as `"PAX"`.

The package in which symbols are read is the Elisp `slime-current-package`. In Lisp buffers, this is the buffer's package, else it's the package of the Slime repl buffer.

### 8.2.2 `M-.` Completion

When `M-.` prompts for the definition to edit, TAB-completion is available in the minibuffer for both names and locatives. To reduce clutter, string names are completed only if they are typed explicitly with an opening quotation mark, and they are case-sensitive. Examples:

- `pri<TAB>` invokes the usual Slime completion.
- `print <TAB>` (note the space) lists `function(0 1)` and `(pax:clhs function)` as locatives.
- `class dref:<TAB>` lists `dref:xref(0 1)` and `dref:dref(0 1)` (all the classes in the package `dref`).
- `pax:locative <TAB>` lists all locative types (see the CL function `dref:locative-types`).
- `package "MGL<TAB>` lists the names of packages that start with `"MGL"`.
- `package <TAB>` lists the names of all packages as strings and also `class`, `mgl-pax:locative` because `package` denotes a class and also a locative.

For more powerful search, see Apropos.

# 9 Generating Documentation

## 9.1 The `document` Function

- **[function]** **document** *documentable &key (stream t) pages (format :plain)*

Write `documentable` in `format` to `stream` diverting some output to `pages`. `format` is one of `:plain`, `:markdown`, `:html` and `:pdf`. `stream` may be a `stream` object, `t` or `nil` as with `cl:format`.

To look up the documentation of the `document` function itself:

```
(document #'document)
```

The same with fancy markup:

```
(document #'document :format :markdown)
```

To document a `section`:

```
(document pax::@pax-manual)
```

To generate the documentation for separate libraries with automatic cross-links:

```
(document (list pax::@pax-manual dref::@dref-manual) :format :markdown)
```

See Utilities for Generating Documentation for more.

Definitions that do not define a first-class object are supported via DRef:

```
(document (dref:locate 'foo 'type))
```

There are quite a few special variables that affect how output is generated, see Codification, Linking to the HyperSpec, Linking to Sections, Link Format and Output Formats.

For the details, see the following sections, starting with `documentable`. Also see Writing Extensions and `document-object*`.

### 9.1.1 `documentable`

The `documentable` argument of `document` may be a single object (e.g. `#'print`), a definition such as `(dref 'print 'function)`, a string, or a nested list of these. More precisely, `documentable` is one of the following:

- *single definition designator*: A `dref` or anything else that is `locate`able. This includes non-`dref` `xref`s and first-class objects such as `function`s. The generated documentation typically includes the definition's `docstring`. See Markdown Output for more.
- *docstring*: A string, in which case it is processed like a docstring in `defsection`. That is, with docstring sanitization, Codification, and Linking.
- *list of documentables*: A nested list of `locate`able objects and docstrings. The objects in it are documented in depth-first order. The structure of the list is otherwise unimportant.

### 9.1.2 Return Values

If `pages` are `nil`, then `document` - like `cl:format` - returns a string (when `stream` is `nil`) else `nil`.

If `pages`, then a list of output designators are returned, one for each non-empty page (to which some output has been written), which are determined as follows.

- The string itself if the output was to a string.
- The stream if the output was to a stream.
- The pathname of the file if the output was to a file.

If the default page given by the `stream` argument of `document` was written to, then its output designator is the first element of the returned list. The rest of the designators correspond to the non-empty pages in the `pages` argument of `document` in that order.

### 9.1.3 `pages`

The `pages` argument of `document` is to create multi-page documents by routing some of the generated output to files, strings or streams. `pages` is a list of page specification elements. A page spec is a property list with keys `:objects`, `:output`, `:uri-fragment`, `:source-uri-fn`, `:header-fn` and `:footer-fn`. `objects` is a list of objects (references are allowed but not required) whose documentation is to be sent to `:output`.

`pages` may look something like this:

```
`((;; The section about SECTIONs and everything below it ...
   :objects (, @sections)
   ;; ... is so boring that it's not worth the disk space, so
   ;; send it to a string.
   :output (nil)
   ;; Explicitly tell other pages not to link to these guys.
   :uri-fragment nil)
  ;; Send the @EXTENSION-API section and everything reachable
  ;; from it ...
  (:objects (, @extension-api)
   ;; ... to build/tmp/pax-extension-api.html.
   :output "build/tmp/pax-extension-api.html"
   ;; However, on the web server html files will be at this
   ;; location relative to some common root, so override the
   ;; default:
   :uri-fragment "doc/dev/pax-extension-api.html"
   ;; Set html page title, stylesheet, charset.
   :header-fn 'write-html-header
   ;; Just close the body.
   :footer-fn 'write-html-footer)
  ;; Catch references that were not reachable from the above. It
  ;; is important for this page spec to be last.
  (:objects (, @pax-manual)
   :output "build/tmp/manual.html"
   ;; Links from the extension api page to the manual page will
   ;; be to ../user/pax-manual#<anchor>, while links going to
   ;; the opposite direction will be to
   ;; ../dev/pax-extension-api.html#<anchor>.
   :uri-fragment "doc/user/pax-manual.html"
   :header-fn 'write-html-header
```

```
  :footer-fn 'write-html-footer))
```

Documentation is initially sent to a default stream (the `stream` argument of `document`), but output is redirected if the thing being currently documented is the `:object` of a `page-spec`.

- `:output` can be a number things:
  - If it's `nil`, then output will be collected in a string.
  - If it's `t`, then output will be sent to `*standard-output*`.
  - If it's a stream, then output will be sent to that stream.
  - If it's a list whose first element is a string or a pathname, then output will be sent to the file denoted by that and the rest of the elements of the list are passed on to `cl:open`. One extra keyword argument is `:ensure-directories-exist`. If it's true, `ensure-directories-exist` will be called on the pathname before it's opened.

  Note that even if `pages` is specified, `stream` acts as a catch all, absorbing the generated documentation for references not claimed by any pages.
- `:header-fn`, if not `nil`, is a function of a single stream argument, which is called just before the first write to the page. Since `:format :html` only generates HTML fragments, this makes it possible to print arbitrary headers, typically setting the title, CSS stylesheet, or charset.
- `:footer-fn` is similar to `:header-fn`, but it's called after the last write to the page. For HTML, it typically just closes the body.
- `:uri-fragment` is a string such as `"doc/manual.html"` that specifies where the page will be deployed on a webserver. It defines how links between pages will look. If it's not specified and `:output` refers to a file, then it defaults to the name of the file. If `:uri-fragment` is `nil`, then no links will be made to or from that page.
- `:source-uri-fn` is a function of a single, `dref` argument. If it returns a value other than `nil`, then it must be a string representing an URI. This affects `*document-mark-up-signatures*` and `*document-fancy-html-navigation*`. Also see `make-git-source-uri-fn`.

### 9.1.4 Package and Readtable

While generating documentation, symbols may be read (e.g. from docstrings) and printed. What values of `*package*` and `*readtable*` are used is determined separately for each definition being documented.

- If the values of `*package*` and `*readtable*` in effect at the time of definition were captured (e.g. by `define-locative-type` and `defsection`), then they are used.
- Else, if the definition has a Home Section (see below), then the home section's `section-package` and `section-readtable` are used.
- Else, if the definition has an argument list, then the package of the first argument that's not external in any package is used.

- Else, if the definition is named by a symbol, then its `symbol-package` is used, and `*readtable*` is set to the standard readtable `(named-readtables:find-readtable :common-lisp)`.
- Else, `*package*` is set to the `cl-user` package and `*readtable*` to the standard readtable.

The values thus determined come into effect after the name itself is printed, for printing of the arglist and the docstring.

```
CL-USER> (pax:document #'foo)
- [function] FOO <!> X Y &KEY (ERROR T)

    Do something with X and Y.
```

In the above, the `<!>` marks the place where `*package*` and `*readtable*` are bound.

**Home Section** The home section of an object is a `section` that contains the object's definition in its `section-entries` or `nil`. In the overwhelming majority of cases there should be at most one containing section.

If there are multiple containing sections, the following apply.

- If the name of the definition is a non-keyword symbol, only those containing sections are considered whose package is closest to the `symbol-package` of the name, where closest is defined as having the longest common prefix between the two `package-name`s.
- If there are multiple sections with equally long matches or the name is not a non-keyword symbol, then it's undefined which one is the home section.

For example, `(mgl-pax:document function)` is an entry in the `MGL-PAX::@BASICS` section. Unless another section that contains it is defined in the MGL-PAX package, the home section is guaranteed to be `MGL-PAX::@BASICS` because the `symbol-package`s of `mgl-pax:document` and `MGL-PAX::@BASICS` are the same (hence their common prefix is maximally long).

This scheme would also work, for example, if the home package of `document` were `mgl-pax/impl`, and it were reexported from `mgl-pax` because the only way to externally change the home package would be to define a containing section in a package like `mgl-pax/imp`.

Thus, relying on the package system makes it possible to find the intended home section of a definition among multiple containing sections with high probability. However, for names which are not symbols, there is no package system to advantage of.

- **[variable]** ***document-normalize-packages*** *t*

  Whether to print `[in package <package-name>]` in the documentation when the package changes.

## 9.2 Browsing Live Documentation

Documentation for definitions in the running Lisp can be browsed directly without generating documentation in the offline manner. HTML documentation, complete with Codification and Linking, is generated from docstrings of all kinds of definitions and PAX `section`s in the running

Lisp on the fly. This allows ad-hoc exploration of the Lisp, much like `describe-function`, `apropos-command` and other online help commands in Emacs, for which direct parallels are provided.

Still, even without Emacs and SLIME, limited functionality can be accessed through PAX Live Home Page by starting the live documentation web server manually.

If Emacs Setup has been done, the Elisp function `mgl-pax-document` (maybe bound to `C-.`) generates and displays documentation as a single HTML page. If necessary, a disambiguation page is generated with the documentation of all matching definitions. For example, to view the documentation of this very `section`, one can do:

```
M-x mgl-pax-document
View Documentation of: pax::@browsing-live-documentation
```

Alternatively, pressing `C-.` with point over the text `pax::@browsing-live-documentation` in a buffer achieves the same effect.

In interactive use, `mgl-pax-document` behaves similarly to `M-.` except:

- It shows the `documentation` of some definition and does not visit its `source-location`.
- It considers definitions with all `locative-types` not just `lisp-locative-types` because it doesn't need `source-location`.

  This also means that completion works for `clhs` definitions:

  - `"lambda list<TAB>` lists `"lambda list"` and `"lambda list keywords"`, both HyperSpec glossary entries. This is similar to `common-lisp-hyperspec-glossary-term` in Elisp but also works for HyperSpec section titles.
  - `"#<tab>` lists all sharpsign reader macros (similar to `common-lisp-hyperspec-lookup-reader-macro` in Elisp).
  - `"~<tab>` lists all `cl:format` directives (similar to `common-lisp-hyperspec-format` in Elisp).
  - `"loop:~<TAB>` lists all loop keywords.
- It works in non-`lisp-mode` buffers by reinterpreting a few lines of text surrounding point as lisp code (hence the suggested *global* binding).
- It supports fragment syntax at the prompt:

  ```
  NAME LOCATIVE FRAGMENT-NAME FRAGMENT-LOCATIVE
  ```

  This is like `name locative`, but the browser scrolls to the definition of `fragment-name fragment-locative` within that page.

  For example, entering this at the prompt will generate the entire PAX manual as a single page and scroll to the very section you are reading within it:

  ```
  pax::@pax-manual pax:section pax::@browsing-live-documentation pax:section
  ```

- If the empty string is entered at the prompt, and there is no existing w3m buffer or w3m is not used, then PAX Live Home Page is visited. If there is a w3m buffer, then entering the empty string displays that buffer.

The convenience function `mgl-pax-current-definition-toggle-view` (`C-c C-d c`) documents the definition with point in it.

### 9.2.1 Browsing with w3m

When the value of the Elisp variable `mgl-pax-browser-function` is `w3m-browse-url` (see Emacs Setup), the Emacs w3m browser is used without the need for a web server, and also offering somewhat tighter integration than Browsing with Other Browsers.

With w3m's default key bindings, moving the cursor between links involves `tab` and `s-tab` (or `<up>` and `<down>`). `ret` and `<right>` follow a link, while `b` and `<left>` go back in history.

In addition, the following PAX-specific key bindings are available:

- `M-.` visits the source location of the definition corresponding to the link under the point.
- Invoking `mgl-pax-document` on a section title link will show the documentation of that section on its own page.
- `n` moves to the next PAX definition on the page.
- `p` moves to the previous PAX definition on the page.
- `u` follows the first `Up:` link (to the first containing `section`) if any.
- `U` is like `u` but positions the cursor at the top of the page.
- `v` visits the source location of the current definition (the one under the cursor or the first one above it).
- `V` visits the source location of the first definition on the page.

### 9.2.2 Browsing with Other Browsers

When the value of the Elisp variable `mgl-pax-browser-function` is not `w3m-browse-url` (see Emacs Setup), requests are served via a web server started in the running Lisp, and documentation is most likely displayed in a separate browser window.

By default, `mgl-pax-browser-function` is `nil`, which makes PAX use `browse-url-browser-function`. You may want to customize the related `browse-url-new-window-flag` or, for Chrome, set `browse-url-chrome-arguments` to `("--new-window")`.

By default, `mgl-pax-web-server-port` is `nil`, and PAX will pick a free port automatically.

In the browser, clicking on the locative on the left of the name (e.g. in `- [function] PRINT`) will raise and focus the Emacs window (if Emacs is not in text mode, and also subject to window manager focus stealing settings), then go to the corresponding source location. For sections, clicking on the lambda link will do the same (see `*document-fancy-html-navigation*`).

Finally, note that the `urls` exposed by the web server are subject to change, and even the port used may vary by session if the Elisp variable `mgl-pax-web-server-port` is nil.

- **[variable]** ***browse-html-style*** *:charter*

  The HTML style to use for browsing live documentation. Affects only non-w3m browsers. See `*document-html-default-style*` for the possible values.

  If you change this variable, you may need to do a hard refresh in the browser (often `C-<f5>`).

### 9.2.3 Apropos

The Elisp functions `mgl-pax-apropos`, `mgl-pax-apropos-all`, and `mgl-pax-apropos-package` can display the results of `dref-apropos` in the live documentation browser. These extend the functionality of `slime-apropos`, `slime-apropos-all` and `slime-apropos-package` to support more kinds of definitions in an extensible way. The correspondence is so close that the PAX versions might take over the Slime key bindings.

Note that apropos functionality is also exposed via the PAX Live Home Page.

More concretely, the PAX versions supports the following extensions:

- Definitions with string names. One can search for `asdf:system`s, `package`s and `clhs` sections, glossary entries, format directives, reader macro characters, loop keywords.
- Exact or substring matching of the name and the package.
- Matching only symbol or string names.

On the PAX Live Home Page, one may Browse by Locative Types, which gives access to some of the apropos functionality via the browser without involving Emacs.

On the result page:

- A `dref-apropos` form to reproduce the results at the REPL is shown.
- One may toggle the `external-only` and `case-sensitive` boolean arguments.
- One may switch between list, and detailed view. The list view only shows the first, bulleted line for each definition, while the detailed view includes the full documentation of definitions with the exception of `section`s.
- The returned references are presented in two groups: those with non-symbol and those with symbol names. The non-symbol group is sorted by locative type then by name. The symbol group is sorted by name then by locative type.

With `mgl-pax-apropos-all` and `mgl-pax-apropos-package` being simple convenience functions on top of `mgl-pax-apropos`, we only discuss the latter in detail here. For the others, see the Elisp docstrings.

**The `string` Argument of `mgl-pax-apropos`** The `string` argument consists of a name pattern and a `dtype`.

The name pattern has the following forms.

- `:print` matches definitions whose names are the string `print` or a symbol with `symbol-name` `print`. Vertical bar form as in `:|prInt|` is also also supported and is useful in when `case-sensitive` is true.
- `"print"` matches definitions whose names contain `print` as a substring.
- `print` is like the previous, substring matching case. Use this form to save typing if the pattern does not contain spaces and does not start with a colon.
- The empty string matches everything.

After the name pattern, `string` may contain a `dtype` that the definitions must match.

- `print t` matches definitions with `lisp-locative-types`, which is the default (equivalent to `print`).
- `print function` matches functions whose names contain `print` (e.g. `cl:print` and `cl:pprint`).
- `:print function` is like the previous example but with exact name match (so it matches `cl:print` but not `cl:pprint`).
- `print variable` matches for example `*print-escape*`.
- `print (or variable function)` matches all variables and functions with `print` in their names.
- `array (or type (not class))` matches `deftype`s and but not `class`es with the string `array` in their names.
- ` pax:section` (note the leading space) matches all PAX sections (`external-only` `nil` is necessary to see many of them).
- `print dref:pseudo` matches definitions with `pseudo-locative-types` such as `mgl-pax:clhs`.
- `print dref:top` matches definitions with all locative types (`locative-types`).

**The `package` Argument of `mgl-pax-apropos`** When `mgl-pax-apropos` is invoked with a prefix argument, it prompts for a package pattern among other things. The pattern may be like the following examples.

- `:none` restricts matches to non-symbol names.
- `:any` restricts matches to symbol names.
- `:cl` restricts matches to symbols in the CL package.
- `:|x y|` is similar to the previous, but the vertical bar syntax allows for spaces in names.
- `mgl` restricts matches to packages whose name contains `mgl` as a substring.
- `"x y"` is the same as the previous, but the explicit quotes allow for spaces in names.

The above examples assume case-insensitive matching.

### 9.2.4 PAX Live Home Page

When Browsing Live Documentation, the home page provides quick access to documentation of the definitions in the system. In Emacs, when `mgl-pax-document` is invoked with the empty string, it visits the home page.

The home page may also be accessed directly by going to the root page of the web server (if one is started). Here, unless the home page is viewed with w3m, one may directly look up documentation and access Apropos via the input boxes provided.

- **[function]** **ensure-web-server** *&key port hyperspec-root*

  Start or update a web server on `port` for Browsing Live Documentation. Returns the base `url` of the server (e.g. `http://localhost:32790`), which goes to the PAX Live Home Page. If the web server is running already `(ensure-web-server)` simply returns its base `url`.

  Note that even when using Emacs but Browsing with Other Browsers, the web server is started automatically. When Browsing with w3m, no web server is involved at all. Calling this function explicitly is only needed if the Emacs integration is not used, or to override `port` and `hyperspec-root`.

  - If `port` is `nil` or 0, then the server will use any free port.
  - If there is a server already running and `port` is not `nil` or 0, then the server is restarted on `port`.
  - If `hyperspec-root` is `nil`, the HyperSpec pages will be served from any previously provided `hyperspec-root` or, failing that, from `*document-hyperspec-root*`.
  - If `hyperspec-root` is non-`nil`, then pages in the HyperSpec will be served from `hyperspec-root`. The following command changes the root without affecting the server in any other way:

    ```
    (ensure-web-server :hyperspec-root "/usr/share/doc/hyperspec/")
    ```

**Top-level PAX Sections** The PAX Live Home Page lists the top-level PAX sections: those that have no other `section`s referencing them (see `defsection`).

**`asdf:system`s and Related `package`s** The PAX Live Home Page lists all `asdf:system`s and `package`s in the Lisp. For easier overview, the they are grouped based on their `source-location`s. Two systems are in the same group if the directory of one (i.e. the directory of the `.asd` file in which it was defined) is the same or is below the other's.

A `package` presented under a group of systems, if the `source-location` of the package is below the the top-most directory among the systems in the group.

**Systemless Packages** The PAX Live Home Page lists `package`s unrelated to any `asdf:system` as systemless.

**Browse by Locative Types** The PAX Live Home Page provides quick links to Apropos result pages for all Basic Locative Types which may have definitions.

- **[glossary-term] related**

  Two definitions are *related* if the directory of one's `source-location`s contains the directory of the other's.

## 9.3 Markdown Support

The Markdown in docstrings is processed with the 3BMD library.

### 9.3.1 Markdown in Docstrings

- Docstrings can be indented in any of the usual styles. PAX normalizes indentation by stripping the longest run of leading spaces common to all non-blank lines except the first. Thus, the following two docstrings are equivalent:

```
(defun foo ()
  "This is
  indented
  differently")

(defun foo ()
  "This is
indented
differently")
```

- When Browsing Live Documentation, the page displayed can be of, say, a single function within what would constitute the offline documentation of a library. Because markdown reference link definitions, for example

```
[Daring Fireball]: http://daringfireball.net/
```

  can be defined anywhere, they wouldn't be resolvable in that case, their use is discouraged. Currently, only reflink definitions in the vicinity of their uses are resolvable. This is left intentionally vague because the specifics are subject to change.

  See `define-glossary-term` for a better alternative to markdown reference links.

Docstrings of definitions which do not have a Home Section and are not `section`s themselves are assumed to have been written with no knowledge of PAX and to conform to markdown only by accident. These docstrings are thus sanitized more aggressively.

- Indentation of what looks like blocks of Lisp code is rounded up to a multiple of 4. More precisely, non-zero indented lines between blank lines or the docstring boundaries are reindented if the first non-space character of the first line is an `(` or a `;` character.
- Special HTML characters `<&` are escaped.
- Furthermore, to reduce the chance of inadvertently introducing a markdown heading, if a line starts with a string of `#` characters, then the first one is automatically escaped. Thus, the following two docstrings are equivalent:

```
The characters #\Space, #\Tab and
#Return are in the whitespace group.

The characters #\Space, #\Tab and
\#Return are in the whitespace group.
```

### 9.3.2 Syntax Highlighting

For syntax highlighting, GitHub's fenced code blocks markdown extension to mark up code blocks with triple backticks is enabled so all you need to do is write:

```
```elisp
(defun foo ())
```
```

to get syntactically marked up HTML output. Copy `src/style.css` from PAX and you are set. The language tag, `elisp` in this example, is optional and defaults to `common-lisp`.

See the documentation of 3BMD and Colorize for the details.

### 9.3.3 MathJax

Displaying pretty mathematics in TeX format is supported via MathJax. It can be done inline with `$` like this:

```
$\int_0^\infty e^{-x^2} dx=\frac{\sqrt{\pi}}{2}$
```

which is displayed as $\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$, or it can be delimited by `$$` like this:

```
$$\int_0^\infty e^{-x^2} dx=\frac{\sqrt{\pi}}{2}$$
```

to get:

$$\int_0^\infty e^{-x^2} dx = \frac{\sqrt{\pi}}{2}$$

MathJax will leave code blocks (including those inline with backticks) alone. Outside code blocks, escape `$` by prefixing it with a backslash to scare MathJax off.

Escaping all those backslashes in TeX fragments embedded in Lisp strings can be a pain. Pythonic String Reader can help with that.

## 9.4 Codification

- **[variable]** ***document-uppercase-is-code*** *t*

  When true, interesting names extracted from codifiable words marked up as code with backticks. For example, this docstring

  ```
  "T PRINT CLASSes SECTION *PACKAGE* MGL-PAX ASDF
  CaMeL Capital"
  ```

is equivalent to this:

```
"`T` `PRINT` `CLASS`es `SECTION` `*PACKAGE*` `MGL-PAX` `ASDF`
CaMel Capital"
```

and renders as

`t` `print` `classes` `section` `mgl-pax` `asdf` CaMel Capital

where the links are added due to `*document-link-code*`.

To suppress codification, add a backslash to the beginning of the a codifiable word or right after the leading `*` if it would otherwise be parsed as markdown emphasis:

```
"\\SECTION *\\PACKAGE*"
```

The number of backslashes is doubled above because that's how the example looks in a docstring. Note that the backslash is discarded even if `*document-uppercase-is-code*` is false.

- **[glossary-term]** **codifiable**

  A word is *codifiable* if

  - it has a single uppercase character (e.g. it's `t`) and no lowercase characters at all, or
  - there is more than one uppercase character and no lowercase characters between them (e.g. `CLASSes`, `nonREADable`, `CLASS-NAMEs` but not `Classes` or `aTe`.

- **[glossary-term]** **interesting**

  A name is *interesting* if

  - it names a symbol external to its package, or
  - it is at least 3 characters long and names an interned symbol, or
  - it names a Local Definition.

  See Package and Readtable.

- **[variable]** ***document-downcase-uppercase-code*** *nil*

  If true, then all Markdown inline code (e.g. ‘code‘, *which renders as* `code`) – including Codification – which has no lowercase characters is downcased in the output. Characters of literal strings in the code may be of any case. If this variable is `:only-in-markup` and the output format does not support markup (e.g. it's `:plain`), then no downcasing is performed. For example,

  ```
  `(PRINT "Hello")`
  ```

  is downcased to

  ```
  `(print "Hello")`
  ```

  because it only contains uppercase characters outside the string. However,

```
  `MiXed "RESULTS"`
```

is not altered because it has lowercase characters.

If the first two characters are backslashes, then no downcasing is performed, in addition to Escaping Autolinking. Use this to mark inline code that's not Lisp.

```
  Press `\\M-.` in Emacs.
```

## 9.5 Linking

PAX supports linking to definitions either with explicit Reflinks or with Autolinks.

When generating offline documentation, only the definitions in `documentable` may be linkable, but when Browsing Live Documentation, everything is linkable as documentation is generated on-demand.

Many examples in this section link to standard Common Lisp definitions. In the offline case, these will link to external URLs, while in the live case to disambiguation pages that list the definition in the running Lisp and in the HyperSpec.

*Invoking `M-.` on `word` or `name` in any of the following examples will disambiguate based on the textual context, determining the locative.* This is because navigation and linking use the same Parsing algorithm, although linking is a bit more strict about trimming, depluralization, and it performs Filtering Links. On the other hand, `M-.` cannot visit the `clhs` references because there are no associated source locations.

### 9.5.1 Reflink

The Markdown reference link syntax `[label][id]` is repurposed for linking to definitions. In the following, we discuss the various forms of reflinks.

**Specific Reflink** *Format:* `[ word ][ locative ]`

The first name in `word` (with depluralization) that forms a valid `dref` with `locative` is determined, and that definition is linked to. If there is no such `dref`, then an `unresolvable-reflink` warning is signalled.

*Examples:*

- `[``eql(0 1)][type]` *renders as* `eql`.
- `[EQL][type]` *renders as* `eql`.

The Markdown link definition (i.e. `type` above) needs no backticks to mark it as code, but here and below, the second example relies on `*document-uppercase-is-code*` being true.

**Specific Reflink with Text** *Format:* `[link text][ name locative ]`

If `name` and `locative` form a valid `dref`, then that definition is linked to with link text `link text`. If there is no such `dref`, then an `unresolvable-reflink` warning is signalled.

In this form, if `name` starts with `#\"`, then it's read as a string, else as a symbol.

*Examples:*

- `[see this][eql type]` *renders as* see this.
- `[see this]["MGL-PAX" package]` *renders as* see this.

#### Unspecific Reflink *Format:* `[ word ][]`

The first name in `word` (with depluralization, symbols only) that has some `definitions` is determined, and those definitions are linked to. If no name with any definition is found, then an `unresolvable-reflink` warning is signalled.

*Examples:*

- single link: `[print][]` *renders as* `print`.
- multiple links: `[eql][]` *renders as* `eql`(`0` `1`).
- no definitions: `[bad-name][]` *renders as* BAD-NAME.

#### Unspecific Reflink with Text *Format:* `[link text][ name ]`

The `definitions` of `name` are determined, and those definitions are linked to. If `name` has no definitions, then an `unresolvable-reflink` warning is signalled.

In this form, if `name` starts with `#\"`, then it's read as a string, else as as symbol.

*Examples:*

- `[see this][print]` *renders as* see this.
- `[see this][eql]` *renders as* see this(`0` `1`).

#### Markdown Reflink *Format:* `[label][id]`

This is a normal Markdown reference link if `id` is not a valid locative.

- `[see this][user-defined]` renders unchanged.

```
(dref:dref 'user-defined 'locative)
.. debugger invoked on LOCATE-ERROR:
..   Could not locate USER-DEFINED LOCATIVE.
..   USER-DEFINED is not a valid locative type or locative alias.
```

```
(document "[see this][user-defined]" :format :markdown)
.. [see this][user-defined]
..
```

Use `urls` with `define-glossary-term` as a better alternative to Markdown reference links (see Markdown in Docstrings).

**Unresolvable Links**

- **[condition]** **unresolvable-reflink** *warning*

  When `document` encounters a Reflink that looks like a PAX construct but has no matching definition, it signals an `unresolvable-reflink` warning.

  - If the `output-reflink` restart is invoked, then no warning is printed and the markdown link is left unchanged. `muffle-warning(0 1)` is equivalent to `output-reflink`.
  - If the `output-label` restart is invoked, then no warning is printed and the markdown link is replaced by its label. For example, `[NONEXISTENT][function]` becomes `nonexistent`.
  - If the warning is not handled, then it is printed to `*error-output*`, and it behaves as if `output-label` was invoked.

- **[function]** **output-reflink** *&optional condition*

  Invoke the `output-reflink` restart. See `unresolvable-reflink`.

- **[function]** **output-label** *&optional condition*

  Invoke the `output-label` restart. See `unresolvable-reflink`.

### 9.5.2 Autolink

Markdown inline code automatically links to the corresponding definitions without having to use Reflinks. This works especially well in conjunction with Codification. The following examples assume that `*document-uppercase-is-code*` is true. If that's not the case, explicit backticks are required on `word` (but not on `locative`).

**Specific Autolink** *Format:* `word locative` or `locative word`

The first name in `word` (with depluralization) that forms a valid `dref` with `locative` is determined, and that definition is linked to. If no such name is found, then Unspecific Autolink is attempted.

*Examples:*

- `PRINT function` *renders as* `print` function.
- `type EQL` *renders as* type `eql`.
- `type EQL function` *renders as* type `eql` function.

If `locative` has spaces, then it needs to be marked up as code, too. For example,

```
DREF-NAME `(reader dref)`
```

*renders as* `dref-name (reader dref)`.

**Unspecific Autolink** *Format:* `word`

The first name in `word` (with depluralization, symbols only) that has some `definitions` is determined, and those definitions are linked to. If no such name is found or the autolink to this name is *suppressed* (see below), then `word` is left unchanged. If a locative is found before or after `word`, then Specific Autolink is tried first.

*Examples:*

- `print` *renders as* `print`.
- `eql` *renders as* `eql(0 1)`.

Unspecific Autolinking is suppressed if the name found has a Local Definition or was linked to before in the same docstring:

- "`My other CAR is also a CAR`" *renders as* "My other `car` is also a `car`".
- "`[COS][] and COS`" *renders as* "`cos` and `cos`".
- "`[EQL][type] and EQL`" *renders as* "`eql` and `eql`".
- "`EQ and the EQ function`" *renders as* "`eq` and the `eq` function".

Unspecific Autolinking to `t` and `nil` is also suppressed (see `*document-link-to-hyperspec*`):

- "`T and NIL`" *renders as* "`t` and `nil`".

As an exception, a single link (be it either a Specific Link or an unambiguous Unspecific Link) to a `section` or `glossary-term` is not suppressed to allow their titles to be displayed properly:

- "`@NAME and @NAME`" *renders as* "name and name".

**Escaping Autolinking** In the common case, when `*document-uppercase-is-code*` is true, prefixing an uppercase word with a backslash prevents it from being codified and thus also prevents Autolinking form kicking in. For example,

```
\DOCUMENT
```

renders as DOCUMENT. If it should be marked up as code but not autolinked, the backslash must be within backticks like this:

```
`\DOCUMENT`
```

This renders as `document`. Alternatively, the `dislocated` or the `argument` locative may be used as in `[DOCUMENT][dislocated]`.

### 9.5.3 Linking to the HyperSpec

- **[variable]** ***document-link-to-hyperspec*** *t*

  If true, consider definitions found in the Common Lisp HyperSpec for linking. For example,

  - `print` *renders as* `print`.

In offline documentation, this would be a link to the hyperspec unless `#'print` in the running Lisp is `documentable`.

When Browsing Live Documentation, everything is linkable, so the generated link will go to a disambiguation page that lists the definition in the Lisp and in the HyperSpec.

Locatives work as expected (see `*document-link-code*`): `find-if` links to `find-if`, `function` links to `function`(`0` `1`), and `[FUNCTION][type]` links to `function`.

Unspecific Autolinking to `t` and `nil` is suppressed. If desired, use Reflinks such as `[t][]` (that links to `t`(`0` `1`)) or `[T][constant]` (that links to `t`).

Note that linking explicitly with the `clhs` locative is not subject to the value of this variable.

- **[variable]** ***document-hyperspec-root*** *"http://www.lispworks.com/documentation/HyperSpec/"*

  A URL of the Common Lisp HyperSpec. The default value is the canonical location. When invoked from Emacs, the Elisp variable `common-lisp-hyperspec-root` is in effect.

### 9.5.4 Linking to Sections

The following variables control how to generate section numbering, table of contents and navigation links.

- **[variable]** ***document-link-sections*** *t*

  When true, HTML anchors and PDF destinations are generated before the headings (e.g. of sections), which allows the table of contents to contain links and also code-like references to sections (like `@foo-manual`) to be translated to links with the `title` being the link text.

- **[variable]** ***document-max-numbering-level*** *3*

  A non-negative integer. In their hierarchy, sections on levels less than this value get numbered in the format of `3.1.2`. Setting it to 0 turns numbering off.

- **[variable]** ***document-max-table-of-contents-level*** *3*

  An integer that determines the depth of the table of contents.

  - If negative, then no table of contents is generated.
  - If non-negative, and there are multiple top-level sections on a page, then they are listed at the top of the page.
  - If positive, then for each top-level section a table of contents is printed after its heading, which includes a nested tree of section titles whose depth is limited by this value.

  If `*document-link-sections*` is true, then the tables will link to the sections.

- **[variable]** ***document-text-navigation*** *nil*

  If true, then before each heading a line is printed with links to the previous, parent and next section. Needs `*document-link-sections*` to be on to work.

- **[variable]** ***document-fancy-html-navigation*** *t*

  If true and the output format is HTML, then headings get a navigation component that consists of links to the previous, parent, next section, a self-link, and a link to the definition in the source code if available (see `:source-uri-fn` in `document`). This component is normally hidden, it is visible only when the mouse is over the heading. Has no effect if `*document-link-sections*` is false.

### 9.5.5 Filtering Links

- **[variable]** ***document-link-code*** *t*

  Whether definitions of things other than `section`s are allowed to be linkable.

- **[glossary-term]** **linkable**

  When a reference is encountered to definition D while processing documentation for some page C, we say that definition D is *linkable* (from C) if

  - D denotes a `section` and `*document-link-sections*` is true, or
  - D does not denote a `section` and `*document-link-code*` is true

  ... and

  - We are Browsing Live Documentation, or
  - D is an external definition (`clhs` or denotes a `glossary-term` with a `url`), or
  - D's page is C, or
  - D's page is relativizable to C.

  In the above,_D's page_ is the last of the pages in the `documentable` to which D's documentation is written (see `:objects` in `pages`), and we say that a page is *relativizable* to another if it is possible to construct a relative link between their `:uri-fragment`s.

**Specific Link** Specific links are those Reflinks and Autolinks that have a single locative and therefore at most a single matching definition. These are Specific Reflink, Specific Reflink with Text and Specific Autolink.

A specific link to a linkable definition produces a link in the output. If the definition is not linkable, then the output will contain only what would otherwise be the link text.

**Unspecific Link** Unspecific links are those Reflinks and Autolinks that do not specify a locative and match all definitions with a name. These are Unspecific Reflink, Unspecific Reflink with Text and Unspecific Autolink.

To make the links predictable and manageable in number, the following steps are taken.

1. Definitions that are not symbol-based (i.e. whose `dref-name` is not a symbol) are filtered out to prevent unrelated `package`s, `asdf:system`s and `clhs` sections from cluttering the documentation without the control provided by importing symbols.

2. All references with `locative-type` `locative` are filtered out.
3. Non-linkable definitions are removed.
4. If the definitions include a `generic-function`, then all definitions with `locative-type` `method`, `accessor`, `reader` and `writer` are removed to avoid linking to a possibly large number of methods.

If at most a single definition remains, then the output is the same as with a Specific Link. If multiple definitions remain, then the link text is output followed by a number of numbered links, one to each definition.

### 9.5.6 Link Format

The following variables control various aspects of links and `urls`.

- **[variable]** ***document-url-versions*** *(2 1)*

  A list of versions of PAX URL formats to support in the generated documentation. The first in the list is used to generate links.

  PAX emits HTML anchors before the documentation of `sections` (see Linking to Sections) and other things (see Linking). For the function `foo`, in the current version (version 2), the anchor is `<a id="MGL-PAX:FOO%20FUNCTION">`, and its URL will end with `#MGL-PAX:FOO%20FUNCTION`.

  *Note that to make the URL independent of whether a symbol is internal or external to their `symbol-package`, single colon is printed where a double colon would be expected. Package and symbol names are both printed verbatim except for escaping colons and spaces with a backslash. For exported symbols with no funny characters, this coincides with how `prin1` would print the symbol, while having the benefit of making the URL independent of the Lisp printer's escaping strategy and producing human-readable output for mixed-case symbols. No such promises are made for non-ASCII characters, and their URLs may change in future versions. Locatives are printed with `prin1`.*

  Version 1 is based on the more strict HTML4 standard and the id of `foo` is `"x-28MGL-PAX-3A-3AFOO-20FUNCTION-29"`. This is supported by GitHub-flavoured Markdown. Version 2 has minimal clutter and is obviously preferred. However, in order not to break external links, by default, both anchors are generated.

  Let's understand the generated Markdown.

```
(defun foo (x))

(document #'foo :format :markdown)
=> ("<a id=\"x-28MGL-PAX-3AFOO-20FUNCTION-29\"></a>
<a id=\"MGL-PAX:FOO%20FUNCTION\"></a>

- [function] **FOO** *X*
")

(let ((*document-url-versions* '(1)))
  (document #'foo :format :markdown))
```

```
=> ("<a id=\"x-28MGL-PAX-3AFOO-20FUNCTION-29\"></a>

- [function] **FOO** *X*
")
```

- **[variable]** ***document-min-link-hash-length*** *4*

  Recall that Markdown reference links (like `[label][id]`) are used for Linking. It is desirable to have ids that are short to maintain legibility of the generated markdown, but also stable to reduce the spurious diffs in the generated documentation, which can be a pain in a version control system.

  Clearly, there is a tradeoff here. This variable controls how many characters of the MD5 sum of the full link id (the reference as a string) are retained. If collisions are found due to the low number of characters, then the length of the hash of the colliding reference is increased.

  This variable has no effect on the HTML generated from markdown, but it can make markdown output more readable.

- **[variable]** ***document-base-url*** *nil*

  When `*document-base-url*` is non-`nil`, this is prepended to all Markdown relative `url`s. It must be a valid `url` without no query and fragment parts (that is, *http://lisp.org/doc/* but not *http://lisp.org/doc?a=1* or *http://lisp.org/doc#fragment*). Note that intra-page links using only `url` fragments (e.g. and explicit HTML links (e.g. `<a href="...">`) in Markdown are not affected.

## 9.6 Local Definition

While documentation is generated for a definition, that definition is considered local. Other local definitions may also be established. Local definitions inform Codification through interesting names and affect Unspecific Autolinking.

```
(defun foo (x)
  "FOO adds one to X."
  (1+ x)
```

In this example, while the docstring of `foo` is being processed, the global definition `(dref 'foo 'function)` is also considered local, which suppresses linking `foo` in the `foo`'s docstring back to its definition. If `foo` has other definitions, Unspecific Autolinking to those is also suppressed.

Furthermore, the purely local definition `(dref 'x 'argument)` is established, causing the argument name `x` to be codified because `x` is now interesting.

See `documenting-reference` and `with-dislocated-names` in Extending `document`.

## 9.7 Overview of Escaping

Let's recap how escaping Codification, downcasing, and Linking works.

- One backslash in front of a word turns codification off. Use this to prevent codification words such as DOCUMENT, which is all uppercase hence codifiable, and it names an exported symbol hence it is interesting.
- One backslash right after an opening backtick turns autolinking off.
- Two backslashes right after an opening backtick turns autolinking and downcasing off. Use this for things that are not Lisp code but which need to be in a monospace font.

In the following examples capital C/D/A letters mark the presence, and a/b/c the absence of codification, downcasing, and autolinking assuming all these features are enabled by `*document-uppercase-is-code*`, `*document-downcase-uppercase-code*`, and `*document-link-code*`.

```
DOCUMENT                => [`document`][1234]    (CDA)
\DOCUMENT               => DOCUMENT              (cda)
`\DOCUMENT`             => `document`            (CDa)
`\\DOCUMENT`            => `DOCUMENT`            (CdA)
[DOCUMENT][]            => [`document`][1234]    (CDA)
[\DOCUMENT][]           => [DOCUMENT][1234]      (cdA)
[`\DOCUMENT`][]         => [`document`][1234]    (CDA) *
[`\\DOCUMENT`][]        => [`DOCUMENT`][1234]    (CdA)
[DOCUMENT][dislocated]  => `document`            (CDa)
```

Note that in the example marked with *, the single backslash, that would normally turn autolinking off, is ignored because it is in an explicit link.

## 9.8 Output Formats

- **[variable]** ***document-mark-up-signatures*** *t*

  When true, some things such as function names and arglists are rendered as bold and italic. In `:html` and `:pdf` output, locative types become links to sources (if `:source-uri-fn` is provided, see `pages`), and the symbol becomes a self-link for your permalinking pleasure.

  For example, a reference is rendered in markdown roughly as:

  ```
    - [function] foo x y
  ```

  With this option on, the above becomes:

  ```
    - [function] **foo** *x y*
  ```

  Also, in HTML `**foo**` will be a link to that very entry and `[function]` may turn into a link to sources.

### 9.8.1 Markdown Output

By default, `dref`s are documented in the following format.

```
- [<locative-type>] <name> <arglist>

    <docstring>
```

The line with the bullet is printed with `documenting-reference`. The docstring is processed with `document-docstring` while Local Definitions established with `with-dislocated-names` are in effect for all variables locally bound in a definition with `arglist`, and `*package*` is bound to the second return value of `docstring`.

With this default format, PAX supports all locative types, but for some Basic Locative Types defined in DRef and the PAX Locatives, special provisions have been made.

- For definitions with a `variable` or `constant` locative, their initform is printed as their arglist. The initform is the `initform` argument of the locative if provided, or the global symbol value of their name. If no `initform` is provided, and the symbol is globally unbound, then no arglist is printed.

When the printed initform is too long, it is truncated.

- Depending of what the `setf` locative refers to, the `arglist` of the setf expander, setf function, or the method signature is printed as with the `method` locative.
- For definitions with a `method` locative, the arglist printed is the method signature, which consists of the locative's `qualifiers` and `specializers` appended.
- For definitions with an `accessor`, `reader` or `writer` locative, the class on which they are specialized is printed as their arglist.
- For definitions with a `structure-accessor` locative, the arglist printed is the locative's `class-name` argument if provided.
- For definitions with a `class` locative, the arglist printed is the list of immediate superclasses with `standard-object`, `condition` and non-exported symbols omitted.
- For definitions with a `structure` locative, the arglist printed is the list of immediate superclasses with `structure-object` and non-exported symbols omitted.
- For definitions with a `condition` locative, the arglist printed is the list of immediate superclasses with `standard-object`, `condition` and non-exported symbols omitted.
- For definitions with a `asdf:system` locative, their most important slots are printed as an unnumbered list.
- For definitions with the `locative` locative type, their `locative-type-direct-supers` and `locative-type-direct-subs` are printed.
- When documentation is being generated for a definition with the `section` locative, a new (sub)section is opened (see `with-heading`), within which documentation for its each of its `section-entries` is generated. A fresh line is printed after all entries except the last.
- For definitions with a `glossary-term` locative, no arglist is printed, and if non-`nil`, `glossary-term-title` is printed as name.
- For definitions with a `go` locative, its `locative-args` are printed as its arglist, along with a redirection message.
- See the `include` locative.

- For definitions with a `clhs` locative, the `locative-args` are printed as the arglist. For `clhs` `section`s, the title is included in the arglist.
- For definitions with an `unknown` locative, the `locative-args` are printed as the arglist. There is no docstring.

### 9.8.2 PDF Output

When invoked with `:format :pdf`, `document` generates Markdown Output and converts it to PDF with Pandoc, which in turn uses LaTeX. Make sure that they are installed.

```
(with-open-file (s "x.pdf" :direction :output :if-exists :supersede
                           :if-does-not-exist :create)
  (pax:document "Hello, World!" :stream s :format :pdf))
```

To see how the output looks like, visit pax-manual-v0.4.1.pdf and dref-manual-v0.4.1.pdf.

PDF output is similar to `*document-html-default-style*` `:charter` without the off-white tint and with coloured instead of underlined links. The latter is because underlining interferes with hyphenation in LaTeX. As in HTML output, locative types link to the respective definition in the sources on GitHub (see `make-git-source-uri-fn`).

Note that linking from one PDF to another is currently not supported due to the lack of consistent support in PDF viewers. Therefore, such links are replaced by their label or the title if any (e.g. of a `section` or `glossary-term`).

The generation of Markdown is subject to the standard variables (again see `document`). The Markdown to PDF conversion can be customized with the following variables.

- **[variable]** ***document-pandoc-program*** *"pandoc"*

  The name of the Pandoc binary. It need not be an absolute pathname as `path` is searched.

- **[variable]** ***document-pandoc-pdf-options*** *(("-V" "papersize=a4") ("-V" "margin-left=1.03in") ("-V" "margin-right=1.03in") ("-V" "margin-top=1.435in") ("-V" "margin-bottom=1.435in") ("-V" "fontfamily=XCharter") ("-V" "fontsize=11pt") ("-V" "colorlinks=true") ("-V" "linkcolor=blue") ("-V" "urlcolor=Maroon") ("-V" "toccolor=blue") "--verbose")*

  The command-line options to invoke `*document-pandoc-program*` with. For ease of manipulation, related options are grouped into sublists, but the entire nested list is flattened to get the list of options to pass to Pandoc. If `--verbose` is specified, then in addition to Pandoc logging LaTeX sources, PAX will log to `*error-output*` the Markdown that it converts to PDF via LaTeX. The Markdown includes `*document-pandoc-pdf-header-includes*` and `*document-pandoc-pdf-metadata-block*`.

- **[variable]** ***document-pandoc-pdf-header-includes*** *"<too messy to include>"*

  LaTeX code (a string) to include in the preamble via `header-includes`.

  The default includes have no configuration knobs. Look at the value to see how to customize it.

- **[variable]** ***document-pandoc-pdf-metadata-block*** ""

  A Pandoc YAML metadata block as a string.

  Concatenate to this string to customize it.

## 9.9 Documentation Generation Implementation Notes

Documentation Generation is supported on ABCL, AllegroCL, CLISP, CCL, CMUCL, ECL and SBCL, but their outputs may differ due to the lack of some introspective capability. SBCL generates complete output. see `arglist`, `docstring` and `source-location` for implementation notes.

In addition, CLISP does not support the ambiguous case of Browsing Live Documentation because the current implementation relies on Swank to list definitions of symbols (as `variable`, `function`, etc), and that simply doesn't work.

## 9.10 Utilities for Generating Documentation

Two convenience functions are provided to serve the common case of having an ASDF system with some readmes and a directory with for the HTML documentation and the default CSS stylesheet.

- **[function]** **update-asdf-system-readmes** *object asdf-system &key (url-versions '(1)) (formats '(:markdown))*

  Convenience function to generate up to two readme files in the directory holding the `asdf-system` definition. `object` is passed on to `document`.

  If `:markdown` is in `formats`, then `README.md` is generated with anchors, links, inline code, and other markup added. Not necessarily the easiest on the eye in an editor but looks good on GitHub.

  If `:plain` is in `formats`, then `README` is generated, which is optimized for reading in text format. It has less cluttery markup and no Autolinking.

  Example usage:

```
(update-asdf-system-readmes @pax-manual :mgl-pax
                            :formats '(:markdown :plain))
```

  Note that `*document-url-versions*` is bound to `url-versions`, which defaults to using the uglier, version 1 style of `url` for the sake of GitHub.

### 9.10.1 HTML Output

- **[function]** **update-asdf-system-html-docs** *sections asdf-system &key pages (target-dir (asdf/system:system-relative-pathname asdf-system "doc/")) (update-css-p t) (style *document-html-default-style*)*

  Generate pretty HTML documentation for a single ASDF system, possibly linking to GitHub. If `update-css-p`, copy the `style` files to `target-dir` (see `*document-html-default-style*`).

Example usage:

```
(update-asdf-system-html-docs @pax-manual :mgl-pax)
```

The same, linking to the sources on GitHub:

```
(update-asdf-system-html-docs
  @pax-manual :mgl-pax
  :pages
  `((:objects (,mgl-pax::@pax-manual)
     :source-uri-fn ,(make-git-source-uri-fn
                      :mgl-pax
                      "https://github.com/melisgl/mgl-pax"))))
```

See the following variables, which control HTML generation.

- **[variable]** ***document-html-default-style*** *:default*

  The HTML style to use. It's either `style` is either `:default` or `:charter`. The `:default` CSS stylesheet relies on the default fonts (sans-serif, serif, monospace), while `:charter` bundles some fonts for a more controlled look.

  The value of this variable affects the default style of `update-asdf-system-html-docs`.

- **[variable]** ***document-html-max-navigation-table-of-contents-level*** *nil*

  `nil` or a non-negative integer. If non-`nil`, it overrides `*document-max-numbering-level*` in the dynamic HTML table of contents on the left of the page.

- **[variable]** ***document-html-lang*** "*en*"

  The value for the `html` element's `xml:lang` and `lang` attributes in the generated HTML.

- **[variable]** ***document-html-charset*** "*UTF-8*"

  The value for `charset` attribute of the `<meta http-equiv='Content-Type' content='text/html'>` element in the generated HTML.

- **[variable]** ***document-html-head*** *nil*

  Stuff to be included in the `<head>` of the generated HTML.

  - If `nil`, nothing is included.
  - If a `string`(`0` `1`), then it is written to the HTML output as is without any escaping.
  - If a function designator, then it is called with a single argument, the HTML stream, where it must write the output.

- **[variable]** ***document-html-sidebar*** *nil*

  Stuff to be included in the HTML sidebar.

  - If `nil`, a default sidebar is generated, with `*document-html-top-blocks-of-links*`, followed by the dynamic table of contents, and `*document-html-bottom-blocks-of-links*`.

- If a `string`(`0 1`), then it is written to the HTML output as is without any escaping.
- If a function designator, then it is called with a single argument, the HTML stream, where it must write the output.

- **[variable]** ***document-html-top-blocks-of-links*** *nil*

  A list of blocks of links to be displayed on the sidebar on the left, above the table of contents. A block is of the form `(&key title id links)`, where `title` will be displayed at the top of the block in a HTML `div` with `id` followed by the links. `links` is a list of `(uri label)` elements, where `uri` maybe a string or an object being `document`ed or a `reference` thereof.

- **[variable]** ***document-html-bottom-blocks-of-links*** *nil*

  Like `*document-html-top-blocks-of-links*`, only it is displayed below the table of contents.

### 9.10.2 GitHub Workflow

It is generally recommended to commit generated readmes (see `update-asdf-system-readmes`), so that users have something to read without reading the code and sites like GitHub can display them.

HTML documentation can also be committed, but there is an issue with that: when linking to the sources (see `make-git-source-uri-fn`), the commit id is in the link. This means that code changes need to be committed first, and only then can HTML documentation be regenerated and committed in a followup commit.

The second issue is that GitHub is not very good at serving HTML files from the repository itself (and http://htmlpreview.github.io chokes on links to the sources).

The recommended workflow is to use gh-pages, which can be made relatively painless with the `git worktree` command. The gist of it is to make the `doc/` directory a checkout of the branch named `gh-pages`. There is a good description of this general process. Two commits are needed still, but it is somewhat less painful.

This way the HTML documentation will be available at

```
http://<username>.github.io/<repo-name>
```

It is probably a good idea to add sections like the Links and Systems section to allow jumping between the repository and the gh-pages site.

- **[function]** **make-github-source-uri-fn** *asdf-system github-uri &key git-version*

  This function is a backward-compatibility wrapper around `make-git-source-uri-fn`, which supersedes `make-github-source-uri-fn`. All arguments are passed on to `make-git-source-uri-fn`, leaving `uri-format-string` at its default, which is suitable for GitHub.

- **[function]** **make-git-source-uri-fn** *asdf-system git-forge-uri &key git-version (uri-format-string "~A/blob/~A/~A#L~S")*

Return an object suitable as `:source-uri-fn` of a page spec (see the `pages` argument of `document`). The function looks at the source location of the object passed to it, and if the location is found, the path is made relative to the top-level directory of the git checkout containing the file of the `asdf-system` and finally an URI pointing to your git forge (such as GitHub) is returned. A warning is signalled whenever the source location lookup fails or if the source location points to a directory not below the directory of `asdf-system`.

If `git-forge-uri` is `"https://github.com/melisgl/mgl-pax/"` and `git-version` is `"master"`, then the returned URI may look like this:

```
https://github.com/melisgl/mgl-pax/blob/master/src/pax-early.lisp#L12
```

If `git-version` is `nil`, then an attempt is made to determine to current commit id from the `.git` in the directory holding `asdf-system`. If no `.git` directory is found, then no links to the git forge will be generated.

`uri-format-string` is a `cl:format` control string for four arguments:

- `git-forge-uri`,
- `git-version`,
- the relative path to the file of the source location of the reference,
- and the line number.

The default value of `uri-format-string` is for GitHub. If using a non-standard git forge, such as Sourcehut or GitLab, simply pass a suitable `uri-format-string` matching the URI scheme of your forge.

### 9.10.3 PAX World

PAX World is a registry of documents, which can generate cross-linked HTML documentation pages for all the registered documents. There is an official PAX World.

- **[function]** **register-doc-in-pax-world** *name sections page-specs*

  Register `sections` and `page-specs` under `name` (a symbol) in PAX World. By default, `update-pax-world` generates documentation for all of these. `sections` and `page-specs` must be lists of `section`s and `page-spec`s (SEE `document`) or designators of function of no arguments that return such lists.

For example, this is how PAX registers itself:

```
(defun pax-sections ()
  (list @pax-manual))

(defun pax-pages ()
  `((:objects ,(pax-sections)
     :source-uri-fn ,(make-git-source-uri-fn
                      :mgl-pax
                      "https://github.com/melisgl/mgl-pax"))))

(register-doc-in-pax-world :pax 'pax-sections 'pax-pages)
```

- **[function]** **update-pax-world** *&key (docs *registered-pax-world-docs*) dir update-css-p (style *document-html-default-style*)*

  Generate HTML documentation for all `docs`. Files are created in `dir` (`(asdf:system-relative-pathname :mgl-pax "world/")` by default if `dir` is `nil`). `docs` is a list of entries of the form (`name` `sections` `page-specs`). The default for `docs` is all the sections and pages registered with `register-doc-in-pax-world`.

  In the absence of `:header-fn` `:footer-fn`, `:output`, every spec in `page-specs` is augmented with HTML headers, footers and output location specifications (based on the name of the section).

  If necessary a default page spec is created for every section.

## 10 Transcripts

What are transcripts for? When writing a tutorial, one often wants to include a REPL session with maybe a few defuns and a couple of forms whose output or return values are shown. Also, in a function's docstring an example call with concrete arguments and return values speaks volumes. A transcript is a text that looks like a REPL session, but which has a light markup for printed output and return values, while no markup (i.e. prompt) for Lisp forms. PAX transcripts may include output and return values of all forms, or only selected ones. In either case, the transcript itself can be easily generated from the source code.

The main worry associated with including examples in the documentation is that they tend to get out-of-sync with the code. This is solved by being able to parse back and update transcripts. In fact, this is exactly what happens during documentation generation with PAX. Code sections tagged with `cl-transcript` are retranscribed and checked for consistency (that is, no difference in output or return values). If the consistency check fails, an error is signalled that includes a reference to the object being documented.

Going beyond documentation, transcript consistency checks can be used for writing simple tests in a very readable form. For example:

```
(+ 1 2)
=> 3

(values (princ :hello) (list 1 2))
.. HELLO
=> :HELLO
=> (1 2)
```

All in all, transcripts are a handy tool especially when combined with the Emacs support to regenerate them and with PYTHONIC-STRING-READER's triple-quoted strings, that allow one to work with nested strings with less noise. The triple-quote syntax can be enabled with:

```
(in-readtable pythonic-string-syntax)
```

### 10.1 Transcribing with Emacs

Typical transcript usage from within Emacs is simple: add a Lisp form to a docstring or comment at any indentation level. Move the cursor right after the end of the form as if you were to evaluate it with `C-x C-e`. The cursor is marked by `#\^`:

```
This is part of a docstring.

```cl-transcript
(values (princ :hello) (list 1 2))^
```
```

Note that the use of fenced code blocks with the language tag `cl-transcript` is only to tell PAX to perform consistency checks at documentation generation time.

Now invoke the Elisp function `mgl-pax-transcribe` where the cursor is, and the fenced code block from the docstring becomes:

```
(values (princ :hello) (list 1 2))
.. HELLO
=> :HELLO
=> (1 2)
^
```

Then you change the printed message and add a comment to the second return value:

```
(values (princ :hello-world) (list 1 2))
.. HELLO
=> :HELLO
=> (1
    ;; This value is arbitrary.
    2)
```

When generating the documentation you get a `transcription-consistency-error` because the printed output and the first return value changed, so you regenerate the documentation by marking the region bounded by `#\|` and the cursor at `#\^` in the example:

```
|(values (princ :hello-world) (list 1 2))
.. HELLO
=> :HELLO
=> (1
    ;; This value is arbitrary.
    2)
^
```

then invoke the Elisp function `mgl-pax-retranscribe-region` to get:

```
(values (princ :hello-world) (list 1 2))
.. HELLO-WORLD
=> :HELLO-WORLD
=> (1
    ;; This value is arbitrary.
    2)
```

```
^
```

Note how the indentation and the comment of `(1 2)` were left alone, but the output and the first return value got updated.

Alternatively, `C-u 1 mgl-pax-transcribe` will emit commented markup:

```
(values (princ :hello) (list 1 2))
;.. HELLO
;=> :HELLO
;=> (1 2)
```

`C-u 0 mgl-pax-retranscribe-region` will turn commented into non-commented markup. In general, the numeric prefix argument is the index of the syntax to be used in `*transcribe-syntaxes*`. Without a prefix argument, `mgl-pax-retranscribe-region` will not change the markup style.

Finally, not only do both functions work at any indentation level but in comments too:

```
;;;; (values (princ :hello) (list 1 2))
;;;; .. HELLO
;;;; => :HELLO
;;;; => (1 2)
```

The dynamic environment of the transcription is determined by the `:dynenv` argument of the enclosing `cl-transcript` code block (see Controlling the Dynamic Environment).

Transcription support in Emacs can be enabled by loading `src/mgl-pax.el`. See Emacs Setup.

## 10.2 Transcript API

- **[function]** **transcribe** *input output &key update-only (include-no-output update-only) (include-no-value update-only) (echo t) (check-consistency *transcribe-check-consistency*) default-syntax (input-syntaxes *transcribe-syntaxes*) (output-syntaxes *transcribe-syntaxes*) dynenv*

  Read forms from `input` and write them (iff `echo`) to `output` followed by any output and return values produced by calling `eval` on the form. The variables `*`, `**`, `***`, `/`, `//`, `///`, `-`, `+`, `++`, `+++` are locally bound and updated as in a REPL. Since `transcribe` `eval`uates arbitrary code anyway, forms are read with `*read-eval*` `t`.

  `input` can be a stream or a string, while `output` can be a stream or `nil`, in which case output goes into a string. The return value is the `output` stream or the string that was constructed.

  Go up to Transcribing with Emacs for nice examples. A more mind-bending one is this:

  ```
  (transcribe "(princ 42) " nil)
  => "(princ 42)
  .. 42
  => 42
  "
  ```

However, the above may be a bit confusing since this documentation uses `transcribe` markup syntax in this very example, so let's do it differently. If we have a file with these contents:

```
(values (princ 42) (list 1 2))
```

it is transcribed to:

```
(values (princ 42) (list 1 2))
.. 42
=> 42
=> (1 2)
```

Output to all standard streams is captured and printed with the `:output` prefix (`".."`). The return values above are printed with the `:readable` prefix (`"=>"`). Note how these prefixes are always printed on a new line to facilitate parsing.

**Updating**

`transcribe` is able to parse its own output. If we transcribe the previous output above, we get it back exactly. However, if we remove all output markers, leave only a placeholder value marker and pass `:update-only t` with source:

```
(values (princ 42) (list 1 2))
=>
```

we get this:

```
(values (princ 42) (list 1 2))
=> 42
=> (1 2)
```

With `update-only`, the printed output of a form is transcribed only if there were output markers in the source. Similarly, with `update-only`, return values are transcribed only if there were value markers in the source.

**No Output/Values**

If the form produces no output or returns no values, then whether or not output and values are transcribed is controlled by `include-no-output` and `include-no-value`, respectively. By default, neither is on so:

```
(values)
..
=>
```

is transcribed to

```
(values)
```

With `update-only` true, we probably wouldn't like to lose those markers since they were put there for a reason. Hence, with `update-only`, `include-no-output` and `include-no-value` default to true. So, with `update-only` the above example is transcribed to:

```
(values)
..
=> ; No value
```

where the last line is the `:no-value` prefix.

### Consistency Checks

If `check-consistency` is true, then `transcribe` signals a continuable `transcription-output-consistency-error` whenever a form's output as a string is different from what was in `input`, provided that `input` contained the output. Similarly, for values, a continuable `transcription-values-consistency-error` is signalled if a value read from the source does not print as the as the value returned by `eval`. This allows readable values to be hand-indented without failing consistency checks:

```
(list 1 2)
=> ;; This is commented, too.
   (1
      ;; Funny indent.
      2)
```

See Transcript Consistency Checking for the full picture.

### Unreadable Values

The above scheme involves `read`, so consistency of unreadable values cannot be treated the same. In fact, unreadable values must even be printed differently for transcribe to be able to read them back:

```
(defclass some-class () ())

(defmethod print-object ((obj some-class) stream)
  (print-unreadable-object (obj stream :type t)
    (format stream "~%~%end")))

(make-instance 'some-class)
==> #<SOME-CLASS
-->
--> end>
```

where `"==>"` is the `:unreadable` prefix and `"-->"` is the `:unreadable-continuation` prefix. As with outputs, a consistency check between an unreadable value from the source and the value from `eval` is performed with `string=` by default. That is, the value from `eval` is printed to a string and compared to the source value. Hence, any change to unreadable values will break consistency checks. This is most troublesome with instances of classes with the default `print-object` method printing the memory address. See Finer-Grained Consistency Checks.

### Errors

If an `error` condition is signalled, the error is printed to the output and no values are returned.

```
(progn
  (print "hello")
  (error "no greeting"))
..
.. "hello"
.. debugger invoked on SIMPLE-ERROR:
..   no greeting
```

To keep the textual representation somewhat likely to be portable, the printing is done with `(format t "#<~S ~S>" (type-of error) (princ-to-string error))`. `simple-conditions` are formatted to strings with `simple-condition-format-control` and `simple-condition-format-arguments`.

**Syntaxes**

Finally, a transcript may employ different syntaxes for the output and values of different forms. When `input` is read, the syntax for each form is determined by trying to match all prefixes from all syntaxes in `input-syntaxes` against a line. If there are no output or values for a form in `input`, then the syntax remains undetermined.

When `output` is written, the prefixes to be used are looked up in `default-syntax` of `output-syntaxes`, if `default-syntax` is not `nil`. If `default-syntax` is `nil`, then the syntax used by the same form in the `input` is used or (if that could not be determined) the syntax of the previous form. If there was no previous form, then the first syntax if `output-syntaxes` is used.

To produce a transcript that's executable Lisp code, use `:default-syntax :commented-1`:

```
(make-instance 'some-class)
;==> #<SOME-CLASS
;-->
;--> end>

(list 1 2)
;=> (1
;->    2)
```

To translate the above to uncommented syntax, use `:default-syntax :default`. If `default-syntax` is `nil` (the default), the same syntax will be used in the output as in the input as much as possible.

**Dynamic Environment**

If `dynenv` is non-`nil`, then it must be a function that establishes the dynamic environment in which transcription shall take place. It is called with a single argument: a thunk (a function of no arguments). See Controlling the Dynamic Environment for an example.

- **[variable]** ***transcribe-check-consistency*** *nil*

  The default value of `transcribe`'s `check-consistency` argument.

- **[variable]** ***transcribe-syntaxes*** *((:default (:output "..") (:no-value "=> ; No value")*

*(:readable "=>") (:unreadable "==>") (:unreadable-continuation "-->")) (:commented-1 (:output ";..") (:no-value ";=> ; No value") (:readable ";=>") (:readable-continuation ";->") (:unreadable ";==>") (:unreadable-continuation ";-->")) (:commented-2 (:output ";;..") (:no-value ";;=> ; No value") (:readable ";;=>") (:readable-continuation ";;->") (:unreadable ";;==>") (:unreadable-continuation ";;-->")))*

The default syntaxes used by `transcribe` for reading and writing lines containing output and values of an evaluated form.

A syntax is a list of of the form `(syntax-id &rest prefixes)` where `prefixes` is a list of `(prefix-id prefix-string)` elements. For example the syntax `:commented-1` looks like this:

```
(:commented-1
 (:output ";..")
 (:no-value ";=>  No value")
 (:readable ";=>")
 (:readable-continuation ";->")
 (:unreadable ";==>")
 (:unreadable-continuation ";-->"))
```

All of the above prefixes must be defined for every syntax except for `:readable-continuation`. If that's missing (as in the `:default` syntax), then the following value is read with `read` and printed with `prin1` (hence no need to mark up the following lines).

When writing, an extra space is added automatically if the line to be prefixed is not empty. Similarly, the first space following the prefix is discarded when reading.

See `transcribe` for how the actual syntax to be used is selected.

- **[condition]** **transcription-error** *error*

  Represents syntactic errors in the `source` argument of `transcribe` and also serves as the superclass of `transcription-consistency-error`.

- **[condition]** **transcription-consistency-error** *transcription-error*

  A common superclass for `transcription-output-consistency-error` and `transcription-values-consistency-error`.

- **[condition]** **transcription-output-consistency-error** *transcription-consistency-error*

  Signalled (with `cerror`) by `transcribe` when invoked with `:check-consistency` and the output of a form is not the same as what was parsed.

- **[condition]** **transcription-values-consistency-error** *transcription-consistency-error*

  Signalled (with `cerror`) by `transcribe` when invoked with `:check-consistency` and the values of a form are inconsistent with their parsed representation.

## 10.3 Transcript Consistency Checking

The main use case for consistency checking is detecting out-of-date examples in documentation, although using it for writing tests is also a possibility. Here, we focus on the former.

When a markdown code block tagged `cl-transcript` is processed during Generating Documentation, the code in it is replaced with the output of with `(transcribe <code> nil :update-only t :check-consistency t)`. Suppose we have the following example of the function `greet`, that prints `hello` and returns 7.

```
```cl-transcript
(greet)
.. hello
=> 7
```
```

Now, if we change `greet` to print or return something else, a `transcription-consistency-error` will be signalled during documentation generation. Then we may fix the documentation or `continue` from the error.

By default, comparisons of previous to current output, readable and unreadable return values are performed with `string=`, `equal`, and `string=`, respectively, which is great in the simple case. Non-determinism aside, exact matching becomes brittle as soon as the notoriously unportable pretty printer is used or when unreadable objects are printed with their `#<>` syntax, especially when `print-unreadable-object` is used with `:identity t`.

### 10.3.1 Finer-Grained Consistency Checks

To get around this problem, consistency checking of output, readable and unreadable values can be customized individually by supplying `transcribe` with a `check-consistency` argument like `((:output <output-check>) (:readable <readable-check>) (:unreadable <unreadable-check>))`. In this case, `<output-check>` may be `nil`, `t`, or a function designator.

- If it's `nil` or there is no `:output` entry in the list, then the output is not checked for consistency.
- If it's `t`, then the outputs are compared with the default, `string=`.
- If it's a function designator, then it's called with two strings and must return whether they are consistent with each other.

The case of `<readable-check>` and `<unreadable-check>` is similar.

Code blocks tagged `cl-transcript` can take arguments, which they pass on to `transcribe`. The following shows how to check only the output.

```
```cl-transcript (:check-consistency ((:output t)))
(error "Oh, no.")
.. debugger invoked on SIMPLE-ERROR:
..   Oh, no.

(make-condition 'simple-error)
==> #<SIMPLE-ERROR {1008A81533}>
```
```

### 10.3.2 Controlling the Dynamic Environment

The dynamic environment in which forms in the transcript are evaluated can be controlled via the `:dynenv` argument of `cl-transcript`.

```
```cl-transcript (:dynenv my-transcript)
...
```
```

In this case, instead of calling `transcribe` directly, the call will be wrapped in a function of no arguments and passed to the function `my-transcript`, which establishes the desired dynamic environment and calls its argument. The following definition of `my-transcript` simply packages up oft-used settings to `transcribe`.

```
(defun my-transcript (fn)
  (let ((*transcribe-check-consistency*
         '((:output my-transcript-output=)
           (:readable equal)
           (:unreadable nil))))
    (funcall fn)))

(defun my-transcript-output= (string1 string2)
  (string= (my-transcript-normalize-output string1)
           (my-transcript-normalize-output string2)))

(defun my-transcript-normalize-output (string)
  (squeeze-whitespace (delete-trailing-whitespace (delete-comments string))))
```

A more involved solution could rebind global variables set in transcripts, unintern symbols created or even create a temporary package for evaluation.

### 10.3.3 Utilities for Consistency Checking

- **[function]** **squeeze-whitespace** *string*

  Replace consecutive whitespace characters with a single space in `string` and trim whitespace from the right. This is useful to undo the effects of pretty printing when building comparison functions for `transcribe`.

- **[function]** **delete-trailing-whitespace** *string*

  Delete whitespace characters after the last non-whitespace character in each line in `string`.

- **[function]** **delete-comments** *string &key (pattern ";")*

  For each line in `string` delete the rest of the line after and including the first occurrence of `pattern`. On changed lines, delete trailing whitespace too. This function does not parse `string` as Lisp forms, hence all occurrences of `pattern` (even those seemingly in string literals) are recognized as comments.

  Let's define a comparison function:

```
(defun string=/no-comments (string1 string2)
  (string= (delete-comments string1) (delete-comments string2)))
```

And use it to check consistency of output:

```
  ```cl-transcript (:check-consistency ((:output string=/no-comments)))
  (format t "hello~%world")
  .. hello     ; This is the first line.
  .. world     ; This is the second line.
  ```
```

Just to make sure the above example works, here it is without being quoted.

```
(format t "hello~%world")
.. hello     ; This is the first line.
.. world     ; This is the second line.
```

# 11 Writing Extensions

## 11.1 Adding New Locatives

Once everything in Extending DRef has been done, there are only a couple of PAX generic functions left to extend.

- **[generic-function]** **document-object*** *object stream*

  Write `object` in `*format*` to `stream`. Specialize this on a subclass of `dref` if that subclass is not `resolve`able, else on the type of object it resolves to. This function is for extension only. Don't call it directly.

- **[generic-function]** **exportable-reference-p** *package symbol locative-type locative-args*

  Return true if `symbol` is to be exported from `package` when it occurs in a `defsection` in a reference with `locative-type` and `locative-args`. `symbol` is accessible in `package`.

  The default method calls `exportable-locative-type-p` with `locative-type` and ignores the other arguments.

  By default, `section`s and `glossary-term`s are not exported although they are `exportable-locative-type-p`. To export symbols naming sections from MGL-PAX, the following method could be added:

  ```
  (defmethod exportable-reference-p ((package (eql (find-package 'mgl-pax)))
                                     symbol (locative-type (eql 'section))
                                     locative-args)
    t)
  ```

- **[generic-function]** **exportable-locative-type-p** *locative-type*

  Return true if symbols in references with `locative-type` are to be exported by default when they occur in a `defsection`. The default method returns `t`, while the methods for

locative types `section`, `glossary-term`, `package`, `asdf:system`, `method` and `include` return `nil`.

This function is called by the default method of `exportable-reference-p` to decide what symbols `defsection` shall export when its `export` argument is true.

Also note that due to the Home Section logic, especially for locative types with string names, `dref-ext:docstring*` should probably return a non-`nil` package.

## 11.2 Locative Aliases

`define-locative-alias` can be used to help `M-.` and Specific Autolinks disambiguate references based on the context of a name as described on Parsing.

The following example shows how to make docstrings read more naturally by defining an alias.

```
(defclass my-string ()
  ())

(defgeneric my-string (obj)
  (:documentation "Convert OBJ to MY-STRING."))

;;; This version of FOO has a harder to read docstring because
;;; it needs to disambiguate the MY-STRING reference.
(defun foo (x)
  "FOO takes and argument X, a [MY-STRING][class] object.")

;;; Define OBJECT as an alias for the CLASS locative.
(define-locative-alias object class)

;;; Note how no explicit link is needed anymore.
(defun foo (x)
  "FOO takes an argument X, a MY-CLASS object.")
```

Similarly, defining the indefinite articles as aliases of the `class` locative can reduce the need for explicit linking.

```
(define-locative-alias a class)
(define-locative-alias an class)
```

Since these are unlikely to be universally helpful, make sure not to export the symbols `a` and `an`.

## 11.3 Extending `document`

For all definitions that it encounters, `document` calls `document-object*` to generate documentation. The following utilities are for writing new `document-object*` methods, which emit markdown.

- **[variable]** ***format***

  Bound by `document` to its `format` argument, this allows markdown output to depend on the output format.

- **[macro]** **with-heading** *(stream object title &key link-title-to) &body body*

  Write a markdown heading with `title` to `stream`. Nested `with-heading`s produce nested headings. If `*document-link-sections*`, generate anchors based on the definition of `object`. `link-title-to` behaves like the `link-title-to` argument of `defsection`.

- **[macro]** **documenting-reference** *(stream &key reference name package readtable (arglist nil)) &body body*

  Write `reference` to `stream` as described in `*document-mark-up-signatures*`, and establish `reference` as a Local Definition for the processing of `body`.

  - `reference` defaults to the reference for which documentation is currently being generated.
  - `name` defaults to `(xref-name reference)` and is printed after the `locative-type`.
  - `*package*` and `*readtable*` are bound to `package` and `readtable` for the duration of printing the `arglist` and the processing of `body`. If either is `nil`, then a default value is computed as described in Package and Readtable.
  - `arglist`:
    - If it is not provided, then it defaults to (`arglist reference`).
    - If `nil`, then it is not printed.
    - If it is a list, then it is must be a lambda list and is printed without the outermost parens and with the package names removed from the argument names.
    - If its is a string, then it must be valid markdown.
  - It is not allowed to have `with-heading` within the dynamic extent of `body`.

- **[macro]** **with-dislocated-names** *names &body body*

  For each name in `names`, establish a Local Definition.

- **[function]** **document-docstring** *docstring stream &key (indentation " ") exclude-first-line-p (paragraphp t)*

  Write `docstring` to `stream`, sanitizing the markdown from it, performing Codification and Linking, finally prefixing each line with `indentation`. The prefix is not added to the first line if `exclude-first-line-p`. If `paragraphp`, then add a newline before and after the output.

- **[function]** **escape-markdown** *string &key (escape-inline t) (escape-html t) (escape-block t)*

  Backslash escape markdown constructs in `string`.

  - If `escape-inline`, then escape the following characters:

    ```
    *_`[]\
    ```

  - If `escape-html`, then escape the following characters:

```
<&
```

    - If `escape-block`, then escape whatever is necessary to avoid starting a new markdown block (e.g. a paragraph, heading, etc).

- **[function]** **prin1-to-markdown** *object &key (escape-inline t) (escape-html t) (escape-block t)*

  Like `prin1-to-string`, but bind `*print-case*` depending on `*document-downcase-uppercase-code*` and `*format*`, and `escape-markdown`.

## 11.4 Sections

`section` objects rarely need to be dissected since `defsection` and `document` cover most needs. However, it is plausible that one wants to subclass them and maybe redefine how they are presented.

- **[class]** **section**

  `defsection` stores its `name`, `title`, `package`, `readtable` and `entries` arguments in `section` objects.

- **[reader]** **section-name** *section (:name)*

  The name of the global variable whose value is this `section` object.

- **[reader]** **section-package** *section (:package)*

  `*package*` will be bound to this package when generating documentation for this section.

- **[reader]** **section-readtable** *section (:readtable)*

  `*readtable*` will be bound to this when generating documentation for this section.

- **[reader]** **section-title** *section (:title)*

  A markdown string or `nil`. Used in generated documentation.

- **[function]** **section-link-title-to** *section*

- **[function]** **section-entries** *section*

  A list of markdown docstrings and `xref`s in the order they occurred in `defsection`.

## 11.5 Glossary Terms

`glossary-term` objects rarely need to be dissected since `define-glossary-term` and `document` cover most needs. However, it is plausible that one wants to subclass them and maybe redefine how they are presented.

- **[class]** **glossary-term**

  See `define-glossary-term`.

- **[reader]** **glossary-term-name** *glossary-term (:name)*

  The name of the global variable whose value is this `glossary-term` object.

- **[reader]** **glossary-term-title** *glossary-term (:title)*

  A markdown string or `nil`. Used in generated documentation (see Markdown Output).

- **[reader]** **glossary-term-url** *glossary-term (:url)*

  A string or `nil`.