

Scribble: PLT Documentation Tool

Version 4.0

June 11, 2008

Scribble is a collection of tools for creating prose documents, especially those that document libraries, and especially for HTML and PDF (via LaTeX) output. More generally, it is useful for cases where you need to deal with Scheme code that is rich in textual content: it has a syntactic extension for writing almost free-form text and a tool for using the scribble syntax for preprocessing text files.

This document itself is written using Scribble. At the time that it was written, its source was available at <http://svn.plt-scheme.org/plt/trunk/collects/scriblings/scribble/> starting with the "scribble.scrbl" file.

Contents

1	How to Scribble Documentation	5
1.1	Getting Started	5
1.2	Document Syntax	6
1.3	Scheme Typesetting and Hyperlinks	7
1.4	Section Hyperlinks	8
1.5	Defining Scheme Bindings	9
1.6	Showing Scheme Examples	11
1.7	Splitting the Document Source	11
1.8	Multi-Page Sections	12
1.9	Style Guide	13
1.9.1	Prose and Terminology	13
1.9.2	Typesetting Code	14
1.9.3	Typesetting Prose	15
1.9.4	Section Titles	15
2	Scribble Layers	16
2.1	Typical Composition	16
2.2	Layer Roadmap	18
3	@-Reader	20
3.1	Concrete Syntax	20
3.1.1	The Command Part	22
3.1.2	The Datum Part	23
3.1.3	The Body Part	24
3.2	Syntax Properties	30

3.3	Interface	31
4	Structures And Processing	34
4.1	Parts	34
4.2	Tags	35
4.3	Collected and Resolved Information	36
4.4	Structure Reference	37
5	Renderer	47
5.1	Base Renderer	47
5.2	Text Renderer	49
5.3	HTML Renderer	49
5.4	Latex Renderer	49
6	Decoding Text	51
7	Document Language	55
8	Document Reader	56
9	Basic Document Forms	57
9.1	Document Structure	57
9.2	Text Styles	59
9.3	Indexing	61
9.4	Tables of Contents	62
10	Scheme	63
11	Manual Forms	64

11.1 Typesetting Code	64
11.2 Documenting Modules	68
11.3 Documenting Forms, Functions, Structure Types, and Values	70
11.4 Documenting Classes and Interfaces	75
11.5 Documenting Signatures	77
11.6 Various String Forms	78
11.7 Links	80
11.8 Indexing	83
11.9 Images	83
11.10Bibliography	84
11.11Miscellaneous	85
11.12Index-Entry Descriptions	86
12 Evaluation and Examples	89
13 BNF Grammars	92
14 Cross-Reference Utilities	94
15 Text Preprocessor	98
15.1 Using External Files	98
Index	100

1 How to Scribble Documentation

Although the `scribble` command-line utility generates output from a Scribble document (run `scribble -h` for more information), documentation of PLT Scheme libraries is normally built by `setup-plt`. This chapter emphasizes the `setup-plt` approach, which more automatically supports links across documents.

1.1 Getting Started

To document a collection or PLaneT package:

- Create a file in your collection or planet package with the file extension `".scribl"`. Beware that the file name you choose will determine the output directory's name. The remainder of these instructions assume that the file is called `"manual.scribl"`.
- Start `"manual.scribl"` like this:

```
#lang scribble/doc
@(require scribble/manual)

@title{My Library}
```

```
Welcome to my documentation: @scheme[(list 'testing 1 2 3)].
```

The first line starts the file in “text” mode, and introduces the `@` syntax to use Scheme bindings. The second line introduces bindings like `title` and `scheme` for writing PLT Scheme documentation. The `title` call (using `@`) produces a title declaration in the text stream.

- Add the following entry to your collect or package's `"info.ss"`:

```
(define scribblings '(("manual.scribl" ())))
```

The `()` above is a list of options. When your document gets large enough that you want it split into multiple pages, add the `'multi-page` option (omitting the quote, since the whole right-hand side of the definition is already quoted).

If you do not already have an `"info.ss"` module, here's a suitable complete module:

```
#lang setup/infotab
(define scribblings '(("manual.scribl" ())))
```

- Run `setup-plt` to build your documentation. For a collection, optionally supply `-l` followed by the collection name to limit the build process to that collection. For a PLaneT package, optionally supply `-P` followed by the package information to limit the build process to that package.

- The generated documentation is normally "doc/manual/index.html" within the collection or PLaneT package directory. If the collection is in PLT Scheme's main "collects" directory, however, then the documentation is generated as "manual/index.html" in the installation's main "doc" directory.

1.2 Document Syntax

Whether in "text" mode or Scheme mode, @ in a document provides an escape to Scheme mode. The syntax of @ is

```
@ <cmd> [ <datum>* ] { <text-body> }
```

where all three parts after @ are optional, but at least one must be present. No spaces are allowed between

- @ and <cmd>, [, or {
- <cmd> and [or {; or
-] and }.

A <cmd> or <datum> is a Scheme datum, while a <text-body> is itself in text mode.

The expansion of @<cmd> into Scheme code is

```
<cmd>
```

When either [] or { } are used, the expansion is

```
(<cmd> <datum>* <parsed-body>*)
```

where <parsed-body>* is the parse result of the <text-body>. The <parsed-body>* part often turns out to be a sequence of Scheme strings.

In practice, the <cmd> is normally a Scheme identifier that is bound to a procedure or syntactic form. If the procedure or form expects further text to typeset, then { } supplies the text. If the form expects other data, typically [] is used to surround Scheme arguments, instead. Sometimes, both [] and { } are used, where the former surround Scheme arguments that precede text to typeset.

Thus,

```
@(require scribble/manual)
@title{My Library}
@scheme[(list 'testing 1 2 3)]
@section[#:tag "here"]{You Are Here}
```

means

```
(require scribble/manual)
(title "My Library")
(scheme (list 'testing 1 2 3))
(section #:tag "here" "You Are Here")
```

For more information on the syntax of @, see §3 “@-Reader”.

In a document that starts #lang scribble/doc, the top level is a text-mode sequence, as the *(text-body)* in a @ form. The parsed sequence is further decoded to turn it into a hierarchy of sections and paragraphs. For example, a linear sequence of section declarations with interleaved text is turned into a list of part instances with all text assigned to a particular part. See §2 “Scribble Layers” for more information on these layers.

1.3 Scheme Typesetting and Hyperlinks

In the document source at the start of this chapter (§1.1 “Getting Started”), the Scheme expression (list 'testing 1 2 3) is typeset properly, but the list identifier is not hyperlinked to the usual definition. To cause list to be hyperlinked, extend the require form like this:

```
(require scribble/manual
         (for-label scheme))
```

This require with for-label declaration introduces a document-time binding for each export of the scheme module. When the document is built, the scheme form detects the binding for list, and so it generates a reference to the specification of list. The setup process detects the reference, and it finds the matching specification in the existing documentation, and ultimately directs the hyperlink to that specification.

Hyperlinks based on for-label and scheme are the preferred mechanism for linking to information outside of a single document. Such links require no information about where and how a binding is documented elsewhere:

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme))
```

```
@title{My Library}
```

```
See also @scheme[list].
```

The scheme form typesets a Scheme expression for inline text, so it ignores the source formatting of the expression. The schemeblock form, in contrast, typesets inset Scheme

code, and it preserves the expression's formatting from the document source.

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme))

@title{My Library}

Some example Scheme code:

@schemeblock[
(define (nobody-understands-me what)
  (list "When I think of all the"
        what
        "I've tried so hard to explain!"))
(nobody-understands-me "glorble snop")
]
```

1.4 Section Hyperlinks

A `section` declaration in a document can include a `#:tag` argument that declares a `hyperlink-target` tag. The `secref` function generates a hyperlink, using the section name as the text of the hyperlink. Use `seclink` to create a hyperlink with text other than the section title.

The following example illustrates section hyperlinks:

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme))

@title{My Library}

Welcome to my documentation: @scheme[(list 'testing 1 2 3)].

@table-of-contents[]

@section[#:tag "chickens"]{Philadelphia Chickens}

Dancing tonight!

@section{Reprise}
```

See `@secref{chickens}`.

Since the page is so short, the hyperlinks in the above example are more effective if you change the "info.ss" file to add the `'multi-file` flag:

```
(define scribblings '("manual.scrbl" (multi-page)))
```

A section can have a tag prefix that applies to all tags as seen from outside the section. Such a prefix is automatically given to each top-level document as processed by `setup-plt`. Thus, referencing a section tag in a different document requires using a prefix, which is based on the target document's main source file. The following example links to a section in the PLT Scheme reference manual:

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme))
@(define ref-src
  '(lib "scribblings/reference/reference.scrbl"))
```

```
@title{My Library}
```

```
See also @italic{@secref[#:doc ref-src]{pairs}}.
```

As mentioned in §1.3 “Scheme Typesetting and Hyperlinks”, however, cross-document references based on `(require (for-label ...))` and `scheme` are usually better than cross-document references using `secref`.

1.5 Defining Scheme Bindings

Use `defproc` to document a procedure, `deform` to document a syntactic form, `defstruct` to document a structure type, etc. These forms provide consistent formatting of definitions, and they declare hyperlink targets for `scheme`-based hyperlinks.

To document a `my-helper` procedure that is exported by "helper.ss" in the "my-lib" collection that contains "manual.scrbl":

- Use `(require (for-label "helper.ss"))` to import the binding information about the bindings of "helper.ss" for use when typesetting identifiers. A relative reference "helper.ss" works since it is relative to the documentation source.
- Add a `@defmodule[my-lib/helper]` declaration, which specifies the library that is being documented within the section. The `defmodule` form needs an absolute module name `mylib/helper`, instead of a relative reference "helper.ss", since the module path given to `defmodule` appears verbatim in the generated documentation.

- Use `defproc` to document the procedure.

Adding these pieces to `"manual.scribl"` gives us the following:

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme
                    "helper.ss"))

@title{My Library}

@defmodule[my-lib/helper]

@defproc[(my-helper [lst list?])
         (listof
          (not/c (one-of/c 'cow)))]{

  Replaces each @scheme['cow] in @scheme[lst] with
  @scheme['aardvark]. }
```

In `defproc`, a contract is specified with each argument to the procedure. In this example, the contract for the `lst` argument is `list?`, which is the contract for a list. After the closing parenthesis that ends the argument sequence, the contract of the result must be given; in this case, `my-helper` guarantees a result that is a list where none of the elements are `'cow`.

Some things to notice in this example and the documentation that it generates:

- The `list?`, `listof`, etc. elements of contracts are hyperlinked to their documentation.
- The result contract is formatted in the generated documentation in the same way as in the source. That is, the source layout of contracts is preserved. (In this case, putting the contract all on one line would be better.)
- In the prose that documents `my-helper`, `lst` is automatically typeset in italic, matching the typesetting in the blue box. The `scheme` form essentially knows that it's used in the scope of a procedure with argument `lst`.
- If you hover the mouse pointer over `my-helper`, a popup reports that it is provided from `my-lib/helper`.
- If you use `my-helper` in any documentation now, as long as that documentation source also has a `(require (for-label ...))` of `"helper.ss"`, then the reference is hyperlinked to the definition above.

See `defproc*`, `defform`, etc. for more information on forms to document Scheme bindings.

1.6 Showing Scheme Examples

The examples form from [scribble/eval](#) helps you generate examples in your documentation. **Warning:** the examples form is especially likely to change or be replaced.

To use examples, the procedures to document must be suitable for use at documentation time; in fact, `examples` uses bindings introduced into the document source by `require`. Thus, to generate examples using `my-helper` from the previous section, `"helper.ss"` must be imported both via `require-for-label` and `require`:

```
#lang scribble/doc
@(require scribble/manual
          scribble/eval ; <--- added
          "helper.ss" ; <--- added
          (for-label scheme
            "helper.ss"))

@title{My Library}

@defmodule[my-lib/helper]{The @schememodname[my-lib/helper]
module---now with extra cows!}

@defproc[(my-helper [lst list?])
         (listof (not/c (one-of/c 'cow)))]{
  Replaces each @scheme['cow] in @scheme[lst] with
  @scheme['aardvark].

  @examples[
    (my-helper '())
    (my-helper '(cows such remarkable cows))
  ]}
}
```

1.7 Splitting the Document Source

In general, a `".scrbl"` file produces a part. A part produced by a document's main source (as specified in the `"info.ss"` file) represents the whole document. The `include-section` procedure can be used to incorporate a part as a sub-part of the enclosing part.

In `"manual.scrbl"`:

```
#lang scribble/doc
@(require scribble/manual)

@title{My Library}
```

```
@defmodule[my-lib/helper]{The @schememodname[my-lib/helper]
module---now with extra cows!}
```

```
@include-section["cows.scrbl"]
@include-section["aardvarks.scrbl"]
```

In "cows.scrbl":

```
#lang scribble/doc
@(require scribble/manual)

@title{Cows}

Wherever they go, it's a quite a show.
```

In "aardvarks.scrbl":

```
#lang scribble/doc
@(require scribble/manual
          (for-label scheme
                    "helper.ss"))

@title{Aardvarks}

@defproc[(my-helper [lst list?])
          (listof (not/c (one-of/c 'cow)))]{

  Replaces each @scheme['cow] in @scheme[lst] with
  @scheme['aardvark].}
```

1.8 Multi-Page Sections

Setting the `'multi-page` option (see §1.4 “Section Hyperlinks”) causes each top-level section of a document to be rendered as a separate HTML page.

To push sub-sections onto separate pages, use the `'toc` style for the enclosing section (as started by `title`, `section`, `subsection`, etc.) and use `local-table-of-contents` to generate hyperlinks to the sub-sections.

Revising "cows.scrbl" from the previous section:

```
#lang scribble/doc
@(require scribble/manual)
```

```

@title[#:style '(toc)]{Cows}

@local-table-of-contents []

@section[#:tag "singing"]{Singing}
Wherever they go, it's a quite a show.

@section{Dancing}
See @secref["singing"].

```

To run this example, remember to change "info.ss" to add the 'multi-page style. You may also want to add a call to `table-of-contents` in "manual.scrbl".

The difference between `table-of-contents` and `local-table-of-contents` is that the latter is ignored for Latex output.

When using `local-table-of-contents`, it often makes sense to include introductory text before the call of `local-table-of-contents`. When the introductory text is less important and when when local table of contents is short, putting the introductory text after the call of `local-table-of-contents` may be appropriate.

1.9 Style Guide

1.9.1 Prose and Terminology

In the descriptive body of `defform`, `defproc`, etc., do not start with "This ...". Instead, start with a sentence whose implicit subject is the form or value being described. Capitalize the first word. Thus, the description will often start with "Returns" or "Produces." Refer to arguments and sub-forms by name.

Do not use the word "argument" to describe a sub-form in a syntactic form; use the term "sub-form" instead, reserving "argument" for values or expressions in a function call. Refer to libraries and languages as such, rather than as "modules" (even though the form to typeset a library or language name is called `schememodname`). Do not call an identifier (i.e., a syntactic element) a "variable" or a "symbol." Do not use the word "expression" for a form that is a definition or might be a definition; use the word "form," instead. Prefer "function" to "procedure."

Avoid cut-and-paste for descriptive text. If two functions are similar, consider documenting them together with `deftogether`. To abstract a description, consider using explicit prose abstraction, such as "x is like y, except that ...," instead of abstracting the source and instantiating it multiple times; often, a prose abstraction is clearer to the reader than a hidden abstraction in the document implementation.

1.9.2 Typesetting Code

Use `id` or a name that ends `-id` in `deform` to mean an identifier, not `identifier`, `variable`, `name`, or `symbol`. Similarly, use `expr` or something that ends `-expr` for an expression position within a syntactic form. Use `body` for a form (definition or expression) in an internal-definition position. Do not use `expr` for something that isn't exactly an expression, `id` for something that isn't exactly an identifier, etc.; instead, use `deform/subs` to define a new non-terminal.

Beware of using `def` together to define multiple variants of a syntactic form or procedure, because each `deform` or `defproc` creates a definition point, but each form or procedure should have a single definition point. (Scribble issues a warning when a binding has multiple definition points.) Instead, use `defproc*` or `deform*`.

Pay attention to the difference between identifiers and meta-variables when using `scheme`, especially outside of `defproc` or `deform`. Prefix a meta-variable with `_`; for example,

```
@scheme[(rator-expr rand-expr ...)]
```

would be the wrong way to refer to the grammar of a function call, because it produces `(rator-expr rand-expr ...)`, where `rator-expr` and `rand-expr` are typeset as variables. The correct description is

```
@scheme[(rator-expr rand-expr ...)]
```

which produces `(rator-expr rand-expr ...)`, where `rator-expr` and `rand-expr` are typeset as meta-variables. The `defproc`, `deform`, etc. forms greatly reduce this burden in descriptions, since they automatically set up meta-variable typesetting for non-literal identifiers. In `deform`, be sure to include literal identifiers (i.e., those not meant as variables, other than the form name being defined) in a `#:literals` clause.

To typeset an identifier with no particular interpretation—syntax, variable, meta-variable, etc.—use `schemeidfont` (e.g., as in `rand-expr` above). Otherwise, use `litchar`, not merely `schemefont` or `verbatim`, to refer to a specific sequence of characters.

When showing example evaluations, use the REPL-snapshot style:

```
@interaction[
(+ 1 2)
]
```

See also the `scribble/eval` library.

Use four dots, `.....`, in place of omitted code, since `....` means repetition.

1.9.3 Typesetting Prose

Refrain from referring to documentation “above” or “below,” and instead have a hyperlink point to the right place.

In prose, use `“` and `”` quotation marks instead of `"`. Use `---` for an em-dash, and do not include spaces on either side, though it will typeset as an en-dash and spaces in HTML output. Use American style for quotation marks and punctuation at the end of quotation marks (i.e., a sentence-terminating period goes inside the quotation marks). Of course, this rule does not apply for quotation marks that are part of code.

Do not use a citation reference (as created by `cite`) as a noun; use it as an annotation.

Do not start a sentence with a Scheme variable name, since it is normally lowercase. For example, use “The *thing* argument is...” instead of “*thing* is...”

1.9.4 Section Titles

Capitalize all words except articles (“the,” “a,” etc.), prepositions, and conjunctions that are not at the start of the title.

A manual title should normally start with a suitable keyword or key phrase (such as “Scribble” for this manual) that is in boldface. If the key word is primarily an executable name, use `exec` instead of `bold`. Optionally add further descriptive text in the title after a colon, where the text starting with the colon is not in boldface.

2 Scribble Layers

Scribble is made of independently usable parts. For example, the Scribble reader can be used in any situation that requires lots of free-form text. You can also skip Scribble's special reader support, and instead use the document-generation structure directly.

2.1 Typical Composition

A Scribble document normally starts

```
#lang scribble/doc
```

Besides making the file a module, this declaration selects the Scribble reader (instead of the usual Scheme reader), and it starts the body of the file in “text” mode. The reader layer mostly leaves text alone, but `@` forms escape to S-expression mode.

A module written as

```
#lang scribble/doc
@(require scribble/manual)

@(define to-be "To Be")

@title{@|to-be| or Not @|to-be|}

@bold{That} is the question.
Whether 'tis nobler...
```

reads as

```
(module <name> scribble/doc
  (require scribble/manual)
  "\n"
  (define to-be "To Be") "\n"
  "\n"
  (title to-be " or Not " to-be) "\n"
  "\n"
  (bold "That") " is the question." "\n"
  "Whether 'tis nobler..." "\n")
```

As shown in this example, the read result is a module whose content mingles text and definitions. The `scribble/doc` language lifts definitions, requires, and provides to the beginning of the module, while everything else is collected into a document bound to the provided identifier `doc`. That is, the module is transformed to something like this:

```

(module <name> scheme/base
  (require scribble/decode
           scribble/manual)
  (define to-be "To Be")
  (define doc
    (decode
     "\n" "\n" "\n"
     (title to-be " or Not " to-be) "\n"
     "\n"
     (bold "That") " is the question." "\n"
     "Whether 'tis nobler..." "\n"))
  (provide doc))

```

The `decode` function produces a `part` structure instance that represents the document. To build the `part` instance, it inspects its arguments to find a `title-decl` value created by `title` to name the part, `part-start` values created by `section` to designate sub-parts, etc.

A `part` is the input to a rendering back-end, such as the HTML renderer. All renderers recognize a fixed structure hierarchy: the content of a part is a *flow*, which is a sequence of *flow elements*, such as paragraphs and tables; a table, in turn, consists of a list of list of flows; a paragraph is a list of *elements*, which can be instances of the `element` structure type, plain strings, or certain special symbols.

The value bound to `doc` in the example above is something like

```

(make-part ....
  (list "To Be" " or Not " "To Be") ; title
  ....
  (make-flow
    (list
      (make-paragraph
        (list (make-element 'bold (list "That"))
              " is the question." "\n"
              "Whether " 'rsquo "tis nobler..."))))
  ....)

```

Notice that the `'` in the input's `'tis` has turned into `'rsquo` (rendered as a curly apostrophe). The conversion to use `'rsquo` was performed by `decode` via `decode-flow` via `decode-paragraph` via `decode-content` via `decode-string`.

In contrast, `(make-element 'bold (list "That"))` was produced by the `bold` function. The `decode` operation is a function, not a syntactic form, and so `bold` has control over its argument before `decode` sees the result. Also, decoding traverses only immediate string arguments.

As it turns out, `bold` also decodes its argument, because the `bold` function is implemented

as

```
(define (bold . strs)
  (make-element 'bold (decode-content strs)))
```

The `verbatim` function, however, does not decode its content, and instead typesets its text arguments directly.

A document module can construct elements directly using `make-element`, but normally functions like `bold` and `verbatim` are used to construct them. In particular, the `scribble/manual` library provides many functions and forms to typeset elements and flow elements.

The `part` structure hierarchy includes built-in element types for setting hyperlink targets and references. Again, this machinery is normally packaged into higher-level functions and forms, such as `secref`, `defproc`, and `scheme`.

2.2 Layer Roadmap

Working roughly from the bottom up, the Scribble layers are:

- `scribble/reader`: A reader that extends the syntax of Scheme with `@`-forms for conveniently embedding a mixin of text and escapes. See §3 “@-Reader”.
- `scribble/struct`: A set of document datatypes and utilities that define the basic layout and processing of a document. For example, the `part` datatype is defined in this layer. See §4 “Structures And Processing”.
- `scribble/base-renderer` with `scribble/html-renderer`, `scribble/latex-renderer`, or `scribble/text-renderer`: A base renderer and mixins that generate documents in various formats from instances of the `scribble/struct` datatypes. See §5 “Renderer”.
- `scribble/decode`: Processes a stream of text, section-start markers, etc. to produce instances of the `scribble/struct` datatypes. See §6 “Decoding Text”.
- `scribble/doclang`: A language to be used for the initial import of a module; processes the module top level through `scribble/decode`, and otherwise provides all of `scheme/base`. See §7 “Document Language”.
- `scribble/doc`: A language that combines `scribble/reader` with `scribble/doclang`. See §8 “Document Reader”.
- `scribble/basic`: A library of basic document operators—such as `title`, `section`, and `secref`—for use with `scribble/decode` and a renderer. See §9 “Basic Document Forms”.

- `scribble/scheme`: A library of functions for typesetting Scheme code. See §10 “Scheme”. These functions are not normally used directly, but instead through `scribble/manual`.
- `scribble/manual`: A library of functions for writing PLT Scheme documentation; re-exports `scribble/basic`. Also, the `scribble/manual-struct` library provides types for index-entry descriptions created by functions in `scribble/manual`. See §11 “Manual Forms”.
- `scribble/eval`: A library of functions for evaluating code at document-build time, especially for showing examples. See §12 “Evaluation and Examples”.
- `scribble/bnf`: A library of support functions for writing grammars. See §13 “BNF Grammars”.
- `scribble/xref`: A library of support functions for using cross-reference information, typically after a document is rendered (e.g., to search). See §14 “Cross-Reference Utilities”.
- `scribble/text`: A language that uses `scribble/reader` preprocessing text files.

The `scribble` command-line utility generates output with a specified renderer. More specifically, the executable installs a renderer, loads the modules specified on the command line, extracts the `doc` export of each module (which must be an instance of `part`), and renders each—potentially with links that span documents.

3 @-Reader

The Scribble @-reader is designed to be a convenient facility for using free-form text in Scheme code, where “@” is chosen as one of the least-used characters in Scheme code.

You can use the reader via MzScheme’s #reader form:

```
#reader(lib "reader.ss" "scribble")@{This is free-form text!}
```

Note that the reader will only read @-forms as S-expressions. The meaning of these S-expressions depends on the rest of your own code.

A PLT Scheme manual more likely starts with

```
#lang scribble/doc
```

which installs a reader, wraps the file content afterward into a MzScheme module, and parses the body into a document using `scribble/decode`. See §8 “Document Reader” for more information.

Another way to use the reader is to use the `use-at-readtable` function to switch the current readtable to a readtable that parses @-forms. You can do this in a single command line:

```
mzscheme -ile scribble/reader "(use-at-readtable)"
```

3.1 Concrete Syntax

Informally, the concrete syntax of @-forms is

```
@ <cmd> [ <datum>* ] { <text-body>* }
```

where all three parts after @ are optional, but at least one should be present. (Note that spaces are not allowed between the three parts.) @ is set as a non-terminating reader macro, so it can be used as usual in Scheme identifiers unless you want to use it as a first character of an identifier; in this case you need to quote with a backslash (`\@foo`) or quote the whole identifier with bars (`|\@foo|`).

```
(define |\@foo| '\@bar@baz)
```

Of course, @ is not treated specially in Scheme strings, character constants, etc.

Roughly, a form matching the above grammar is read as

```
(<cmd>  
 <datum>*)
```

<parsed-body>)*

where *<parsed-body>* is the translation of each *<text-body>* in the input. Thus, the initial *<cmd>* determines the Scheme code that the input is translated into. The common case is when *<cmd>* is a Scheme identifier, which generates a plain Scheme form.

A *<text-body>* is made of text, newlines, and nested @-forms. Note that the syntax for @-forms is the same in a *<text-body>* context as in a Scheme context. A *<text-body>* that isn't an @-form is converted to a string expression for its *<parsed-body>*, and newlines are converted to `"\n"` expressions.

<code>@foo{bar baz blah}</code>	reads as	<code>(foo "bar baz" "\n" "blah")</code>
<code>@foo{bar @baz[3] blah}</code>	reads as	<code>(foo "bar " (baz 3) "\n" "blah")</code>
<code>@foo{bar @baz{3} blah}</code>	reads as	<code>(foo "bar " (baz "3") "\n" "blah")</code>
<code>@foo{bar @baz[2 3]{4 5} blah}</code>	reads as	<code>(foo "bar " (baz 2 3 "4 5") "\n" "blah")</code>

Note that spaces are not allowed before a `[` or a `{`, or they will be part of the following text (or Scheme code). (More on using braces in body texts below.)

```
@foo{bar @baz[2 3] {4 5}} reads as (foo "bar " (baz 2 3) " {4 5}")
```

When the above @-forms appear in a Scheme expression context, the lexical environment must provide bindings for `foo` (as a procedure or a macro).

```
> (let* ([formatter (lambda (fmt)
  (lambda args (format fmt (apply string-append args))))]
 [bf (formatter "~a*")]
 [it (formatter "/~a/")]
 [ul (formatter "_~a_")]
 [text string-append])
 @text{@it{Note}: @bf{This is @ul{not} a pipe}.})
"/Note/: *This is _not_ a pipe*."
```

If you want to see the expression that is actually being read, you can use Scheme's quote.

```
'@foo{bar} reads as '(foo "bar")
```

3.1.1 The Command Part

Besides being a Scheme identifier, the *<cmd>* part of an @-form can have Scheme punctuation prefixes, which will end up wrapping the *whole* expression.

```
@',@foo{blah} reads as ' ',@(foo "blah")
```

When writing Scheme code, this means that `@',@foo{blah}` is exactly the same as `'@',@foo{blah}` and `' ',@@foo{blah}`, but unlike the latter two, the first construct can appear in body texts with the same meaning, whereas the other two would not work (see below).

After the optional punctuation prefix, the *<cmd>* itself is not limited to identifiers; it can be *any* Scheme expression.

```
@(lambda (x) x){blah} reads as ((lambda (x) x) "blah")  
@'(unquote foo){blah} reads as '(,foo "blah")
```

In addition, the command can be omitted altogether, which will omit it from the translation, resulting in an S-expression that usually contains, say, just strings:

```
@{foo bar baz} reads as ("foo bar" "\n" "baz")  
@' {foo bar baz} reads as ' ("foo bar" "\n" "baz")
```

If the command part begins with a `;` (with no newline between the `@` and the `;`), then the construct is a comment. There are two comment forms, one for arbitrary-text and possibly nested comments, and another one for line comments:

```
@; { <any>* }  
@; <anything-else-without-newline>*
```

In the first form, the commented body must still parse correctly; see the description of the body syntax below. In the second form, all text from the `@;` to the end of the line *and* all following spaces (or tabs) are part of the comment (similar to `%` comments in TeX).

```
@foo{bar @; comment baz@;  
      baz@;  
      blah}
```

Tip: if you're editing in a Scheme-aware editor (like DrScheme or Emacs), it is useful to comment out blocks like this:

```
@; {  
  ...
```

```
;}
```

so the editor does not treat the file as having unbalanced parenthesis.

If only the *<cmd>* part of an @-form is specified, then the result is the command part only, without an extra set of parenthesis. This makes it suitable for Scheme escapes in body texts. (More on this below, in the description of the body part.)

```
@foo{x @y z}      reads as (foo "x " y " z")
@foo{x @(* y 2) z} reads as (foo "x " (* y 2) " z")
@{@foo bar}      reads as (foo " bar")
```

Finally, note that there are currently no special rules for using @ in the command itself, which can lead to things like:

```
@@foo{bar}{baz} reads as ((foo "bar") "baz")
```

3.1.2 The Datum Part

The datum part can contains arbitrary Scheme expressions, which are simply stacked before the body text arguments:

```
@foo[1 (* 2 3)]{bar} reads as (foo 1 (* 2 3) "bar")
@foo[@bar{...}]{blah} reads as (foo (bar "...") "blah")
```

The body part can still be omitted, which is essentially an alternative syntax for plain (non-textual) S-expressions:

```
@foo[bar] reads as (foo bar)
@foo{bar @f[x] baz} reads as (foo "bar " (f x) " baz")
```

The datum part can be empty, which makes no difference, except when the body is omitted. It is more common, however, to use an empty body for the same purpose.

```
@foo[] {bar} reads as (foo "bar")
@foo[] reads as (foo)
@foo reads as foo
@foo{} reads as (foo)
```

The most common use of the datum part is for Scheme forms that expect keyword-value arguments that precede the body of text arguments.

```
@foo[#:style 'big]{bar} reads as (foo #:style 'big "bar")
```

3.1.3 The Body Part

The syntax of the body part is intended to be as convenient as possible for free text. It can contain almost any text—the only characters with special meaning is `@` for sub-`@`-forms, and `}` for the end of the text. In addition, a `{` is allowed as part of the text, and it makes the matching `}` be part of the text too—so balanced braces are valid text.

```
@foo{f{o}} reads as (foo "f{o}o")
@foo{{{}}}} reads as (foo "{{}}}")
```

As described above, the text turns to a sequence of string arguments for the resulting form. Spaces at the beginning and end of lines are discarded, and newlines turn to individual `"\n"` strings (i.e., they are not merged with other body parts); see also the information about newlines and indentation below. Spaces are *not* discarded if they appear after the open `{` (before the closing `}`) when there is also text that follows (precedes) it; specifically, they are preserved in a single-line body.

```
@foo{bar} reads as (foo "bar")
@foo{ bar } reads as (foo " bar ")
@foo[1]{ bar } reads as (foo 1 " bar ")
```

If `@` appears in a body, then it is interpreted as Scheme code, which means that the `@`-reader is applied recursively, and the resulting syntax appears as part of the S-expression, among other string contents.

```
@foo{a @bar{b} c} reads as (foo "a " (bar "b") " c")
```

If the nested `@` construct has only a command—no body or datum parts—it will not appear in a subform. Given that the command part can be any Scheme expression, this makes `@` a general escape to arbitrary Scheme code.

```
@foo{a @bar c} reads as (foo "a " bar " c")
@foo{a @(bar 2) c} reads as (foo "a " (bar 2) " c")
```

This is particularly useful with strings, which can be used to include arbitrary text.

```
@foo{A @"} marks the end} reads as (foo "A } marks the end")
```

Note that the escaped string is (intentionally) merged with the rest of the text. This works for `@` too:

```
@foo{The prefix: @"@.} reads as (foo "The prefix: @.")
@foo{@"@x{y}" --> (x "y")} reads as (foo "@x{y} --> (x \"y\")")
```

Alternative Body Syntax

In addition to the above, there is an alternative syntax for the body, one that specifies a new

marker for its end: use `|{` for the opening marker to have the text terminated by a `|}`.

```
@foo|{...}| reads as (foo "...")
@foo|{"}" follows "{"| reads as (foo "\"}\" \" follows \"{ \"")
@foo|{Nesting |{is}| ok}| reads as (foo "Nesting |{is}| ok")
```

This applies to sub-@-forms too—the @ must be prefixed with a |:

```
@foo|{Maze reads as (foo "Maze" "\n"
|@bar{is} (bar "is") "\n"
Life!}| "Life!")
@t|{In |@i|{sub|@"'s}| too}| reads as (t "In " (i "sub@s") " too")
```

Note that the subform uses its own delimiters, `{...}` or `|{...}|`. This means that you can copy and paste Scribble text with @-forms freely, just prefix the @ if the immediate surrounding text has a prefix.

For even better control, you can add characters in the opening delimiter, between the | and the {. Characters that are put there (non alphanumeric ASCII characters only, excluding { and @) should also be used for sub-@-forms, and the end-of-body marker should have these characters in reverse order with paren-like characters ((, [, <) mirrored.

```
@foo|<<<@x{foo} |@{bar}|.|>>>| reads as (foo "@x{foo} |@{bar}|.|")
@foo|!!{X |!!@b{Y}...}!!! reads as (foo "X " (b "Y") "...")
```

Finally, remember that you can use an expression escape with a Scheme string for confusing situations. This works well when you only need to quote short pieces, and the above works well when you have larger multi-line body texts.

Scheme Expression Escapes

In some cases, you may want to use a Scheme identifier (or a number or a boolean etc.) in a position that touches the following text; in these situations you should surround the escaped Scheme expression by a pair of | characters. The text inside the bars is parsed as a Scheme expression.

```
@foo{foo@bar.} reads as (foo "foo" bar.)
@foo{foo@|bar|.} reads as (foo "foo" bar ".")
@foo{foo@3.} reads as (foo "foo" 3.0)
@foo{foo@|3|.} reads as (foo "foo" 3 ".")
```

This form is a generic Scheme expression escape, there is no body text or datum part when you use this form.

```
@foo{foo@|(f 1)|{bar}} reads as (foo "foo" (f 1) "{bar}")
@foo{foo@|bar|[1]{baz}} reads as (foo "foo" bar "[1]{baz}")
```

This works for string expressions too, but note that unlike the above, the string is (intentionally) not merged with the rest of the text:

```
@foo{x@y"z}    reads as (foo "xyz")
@foo{x@|y"z}   reads as (foo "x" "y" "z")
```

Expression escapes also work with *any* number of expressions,

```
@foo{x@|1 (+ 2 3) 4|y} reads as (foo "x" 1 (+ 2 3) 4 "y")

@foo{x@|*          reads as (foo "x" *
      *|y}          * "y")
```

It seems that `@||` has no purpose—but remember that these escapes are never merged with the surrounding text, which can be useful when you want to control the sub expressions in the form.

```
@foo{Alice@||Bob@| reads as (foo "Alice" "Bob"
      |Carol}              "Carol")
```

Note that `@|{...}|` can be parsed as either an escape expression or as the Scheme command part of a `@`-form. The latter is used in this case (since there is little point in Scheme code that uses braces).

```
@|{blah}| reads as ("blah")
```

Comments

As noted above, there are two kinds of Scribble comments: `@;{...}` is a (nestable) comment for a whole body of text (following the same rules for `@`-forms), and `@;...` is a line-comment.

```
@foo{First line@;{there is still a reads as (foo "First line"
      newline here;}          "\n"
      Second line}          "Second line")
```

One useful property of line-comments is that they continue to the end of the line *and* all following spaces (or tabs). Using this, you can get further control of the subforms.

```
@foo{A long @;    reads as (foo "A long single-string arg.")
      single-@;
      string arg.}
```

Note how this is different from using `@||`s in that strings around it are not merged.

Spaces, Newlines, and Indentation

The Scribble syntax treats spaces and newlines in a special way is meant to be sensible for

dealing with text. As mentioned above, spaces at the beginning and end of body lines are discarded, except for spaces between a `{` and text, or between text and a `}`.

```
@foo{bar}    reads as (foo "bar")
@foo{ bar }  reads as (foo " bar ")
@foo{ bar
  baz }      reads as (foo " bar" "\n"
                  "baz ")
```

A single newline that follows an open brace or precedes a closing brace is discarded, unless there are only newlines in the body; other newlines are read as a `"\n"` string

```
@foo{bar
}           reads as (foo "bar")
@foo{
  bar      reads as (foo
                  "bar")
}
@foo{
  bar      reads as (foo
                  "\n"
                  "bar" "\n")
}
@foo{
  bar      reads as (foo
                  "bar" "\n"
                  "\n"
                  "baz")
}
@foo{
}           reads as (foo "\n")
@foo{
}           reads as (foo "\n"
                  "\n")
@foo{ bar
  baz }     reads as (foo " bar" "\n"
                  "baz ")
```

In the parsed S-expression syntax, a single newline string is used for all newlines; you can use `eq?` to identify this line. This can be used to identify newlines in the original `<text-body>`.

```
> (let ([nl (car @' {
                  } )])
```

```

(for-each (lambda (x) (display (if (eq? x nl) "\n..." x)))
  @' {foo
    @,@(list "bar" "\n" "baz")
    blah}})
(newline))
foo
... bar
baz
... blah

```

Spaces at the beginning of body lines do not appear in the resulting S-expressions, but the column of each line is noticed, and all-space indentation strings are added so the result has the same indentation. A indentation string is added to each line according to its distance from the leftmost syntax object (except for empty lines). (Note: if you try these examples on a mzscheme REPL, you should be aware that the reader does not know about the ">" prompt.)

```

@foo{      reads as (foo
  bar          "bar" "\n"
  baz          "baz" "\n"
  blah        "blah")
}

@foo{      reads as (foo
  begin        "begin" "\n" " "
  x++;        "x++;" "\n"
  end}        "end")

@foo{      reads as (foo " "
  a            "a" "\n" " "
  b            "b" "\n"
  c}          "c")

```

If the first string came from the opening { line, it is not prepended with an indentation (but it can affect the leftmost syntax object used for indentation). This makes sense when formatting structured code as well as text (see the last example in the following block).

```

@foo{bar    reads as (foo "bar" "\n" " "
  baz          "baz" "\n"
  bbb}        "bbb")

@foo{ bar   reads as (foo " bar" "\n" " "
  baz          "baz" "\n" " "
  bbb}        "bbb")

```

```

@foo{bar          reads as (foo "bar" "\n"
  baz              "baz" "\n"
  bbb}            "bbb")

@foo{ bar        reads as (foo " bar" "\n"
  baz              "baz" "\n"
  bbb}            "bbb")

@foo{ bar        reads as (foo " bar" "\n"
  baz              "baz" "\n" " "
  bbb}            "bbb")

@text{Some @b{bold reads as (text "Some " (b "bold" "\n"
  text}, and           "text"), and" "\n"
  more text.}         "more text.")

```

Note that each @-form is parsed to an S-expression that has its own indentation. This means that Scribble source can be indented like code, but if indentation matters then you may need to apply indentation of the outer item to all lines of the inner one. For example, in

```

@code{
  begin
    i = 1, r = 1
    @bold{while i < n do
      r *= i++
    done}
  end
}

```

a formatter will need to apply the 2-space indentation to the rendering of the `bold` body.

Note that to get a first-line text to be counted as a leftmost line, line and column accounting should be on for the input port (`use-at-readtable` turns them on for the current input port). Without this,

```

@foo{x1
  x2
  x3}

```

will not have 2-space indentations in the parsed S-expression if source accounting is not on, but

```

@foo{x1
  x2
  x3}

```

will (due to the last line). Pay attention to this, as it can be a problem with Scheme code, for

example:

```
@code{(define (foo x)
      (+ x 1))}
```

For rare situations where spaces at the beginning (or end) of lines matter, you can begin (or end) a line with a `'''`.

```
@foo{          reads as (foo
  @|| bar @||   " bar " "\n"
  @|| baz}      " baz")
```

3.2 Syntax Properties

The Scribble reader attaches properties to syntax objects. These properties might be useful in some rare situations.

Forms that Scribble reads are marked with a `'scribble` property, and a value of a list of three elements: the first is `'form`, the second is the number of items that were read from the datum part, and the third is the number of items in the body part (strings, sub-forms, and escapes). In both cases, a `0` means an empty datum/body part, and `#f` means that the corresponding part was omitted. If the form has neither parts, the property is not attached to the result. This property can be used to give different meanings to expressions from the datum and the body parts, for example, implicitly quoted keywords:

```
(define-syntax (foo stx)
  (let ([p (syntax-property stx 'scribble)])
    (printf ">>> ~s\n" (syntax->datum stx))
    (syntax-case stx ()
      [(_ x ...)
       (and (pair? p) (eq? (car p) 'form) (even? (cadr p)))
       (let loop ([n (/ (cadr p) 2)]
                  [as '()])
         [xs (syntax->list #'(x ...))])
         (if (zero? n)
             (with-syntax ([attrs (reverse as)]
                           [(x ...) xs])
               #'(list 'foo 'attrs x ...))
             (loop (sub1 n)
                    (cons (with-syntax ([key (car xs)]
                                         [val (cadr xs)])
                              #'(key ,val))
                          as)
                      (caddr xs)))))))))
```

```

> @foo[x 1 y (* 2 3)]{blah}
>>> (foo x 1 y (* 2 3) "blah")
(foo ((x 1) (y 6)) "blah")

```

In addition, the Scribble parser uses syntax properties to mark syntax items that are not physically in the original source — indentation spaces and newlines. Both of these will have a `'scribble` property; an indentation string of spaces will have `'indentation` as the value of the property, and a newline will have a `'(newline S)` value where `S` is the original newline string including spaces that precede and follow it (which includes the indentation for the following item). This can be used to implement a verbatim environment: drop indentation strings, and use the original source strings instead of the single-newline string. Here is an example of this.

```

(define-syntax (verb stx)
  (syntax-case stx ()
    [(_ cmd item ...)
     #'(cmd
        #,@(let loop ([items (syntax->list #'(item ...))])
              (if (null? items)
                  '()
                  (let* ([fst (car items)]
                        [prop (syntax-property fst 'scribble)]
                        [rst (loop (cdr items))])
                    (cond [(eq? prop 'indentation) rst]
                          [(not (and (pair? prop)
                                      (eq? (car prop) 'newline)))
                           (cons fst rst)]
                          [else (cons (datum->syntax-object
                                       fst (cadr prop) fst)
                                       rst]])))))))]))

> @verb[string-append]{
  foo
  bar
}
"foo\n bar"

```

3.3 Interface

```
(require scribble/reader)
```

The `scribble/reader` module provides direct Scribble reader functionality for advanced needs.

```
(read [in]) → any
  in : input-port? = (current-input-port)
```

```
(read-syntax [source-name in]) → (or/c syntax? eof-object?)
  source-name : any/c = (object-name in)
  in : input-port? = (current-input-port)
```

These procedures implement the Scribble reader. They do so by constructing a reader table based on the current one, and using that in reading.

```
(read-inside [in]) → any
  in : input-port? = (current-input-port)
```

```
(read-syntax-inside [source-name in]) → (or/c syntax? eof-object?)
  source-name : any/c = (object-name in)
  in : input-port? = (current-input-port)
```

These `-inside` variants parse as if starting inside a `@{...}`, and they return a (syntactic) list. Useful for implementing languages that are textual by default (see "docreader.ss" for example).

```
(make-at-readtable [#:readtable readtable
                  #:command-char command-char
                  #:start-inside? start-inside?
                  #:datum-readtable datum-readtable
                  #:syntax-post-processor syntax-post-proc])
→ readtable?
  readtable : readtable? = (current-readtable)
  command-char : character? = #\@
  start-inside? : any/c = #f
  datum-readtable : (or/c readtable? boolean?          = #t
                    (readtable? . -> . readtable?))
  syntax-post-proc : (syntax? . -> . syntax?) = values
```

Constructs an `@-readtable`. The keyword arguments can customize the resulting reader in several ways:

- `readtable` — a readtable to base the `@-readtable` on.
- `command-char` — the character used for `@-forms`.
- `datum-readtable` — determines the readtable used for reading the datum part. A `#t` values uses the `@-readtable`, otherwise it can be a readtable, or a readtable-to-

readtable function that will construct one from the @-readtable. The idea is that you may want to have completely different uses for the datum part, for example, introducing a convenient `key=val` syntax for attributes.

- `syntax-post-processor` — function that is applied on each resulting syntax value after it has been parsed (but before it is wrapped quoting punctuations). You can use this to further control uses of @-forms, for example, making the command be the head of a list:

```
(use-at-readtable
 #:syntax-post-processor
 (lambda (stx)
  (syntax-case stx ()
    [(cmd rest ...) #'(list 'cmd rest ...)]
    [else (error "@ forms must have a body")]))))
```

- `start-inside?` — if true, creates a readtable for use starting in text mode, instead of S-expression mode.

`(use-at-readtable ...)` → `void?`

Passes all arguments to `make-at-readtable`, and installs the resulting readtable using `current-readtable`. It also enables line counting for the current input-port via `port-count-lines!`.

4 Structures And Processing

(require scribble/struct)

A document is represented as a part, as described in §4.1 “Parts”. This representation is intended to be independent of its eventual rendering, and it is intended to be immutable; rendering extensions and specific data in a document can collude arbitrarily, however.

A document is processed in three passes. The first pass is the *collect pass*, which globally collects information in the document, such as targets for hyperlinking. The second pass is the *resolve pass*, which matches hyperlink references with targets and expands delayed elements (where the expansion should not contribute new hyperlink targets). The final pass is the *render pass*, which generates the resulting document. None of the passes mutate the document, but instead collect information in side `collect-info` and `resolve-info` tables.

4.1 Parts

A *part* is an instance of `part`; among other things, it has a title content, an initial flow, and a list of subsection parts. An `unnumbered-part` is the same as a `part`, but it isn’t numbered. A `versioned-part` is add a version field to `part`. There’s no difference between a part and a full document; a particular source module just as easily defines a subsection (incorporated via `include-section`) as a document.

A *flow* is an instance of `flow`; it has a list of blocks.

A *block* is either a table, an itemization, blockquote, paragraph, or a delayed block.

- A *table* is an instance of `table`; it has a list of list of flows with a particular style. In Latex output, each table cell is typeset as a single line.
- A *itemization* is an instance of `itemization`; it has a list of flows.
- A *blockquote* is an instance of `blockquote`; it has list of blocks that are indented according to a specified style.
- A *paragraph* is an instance of `paragraph`; it has a *content*, which is a list of elements:
 - An *element* can be a string, one of a few symbols, an instance of `element` (possibly `link-element`, etc.), a part-relative element, a delayed element, or anything else allowed by the current renderer.
 - * A string element is included in the result document verbatim, except for space, and unless the element’s style is `'hspace`. In a style other than `'hspace`, consecutive spaces in the output may be collapsed together or replaced with a line break. In the style `'hspace`, all text is converted to uncollapsible spaces that cannot be broken across lines.

- * A symbol element is either `'mdash`, `'ndash`, `'ldquo`, `'lsquo`, `'rsquo`, `'rarr`, or `'prime`; it is rendered as the corresponding HTML entity (even for Latex output).
 - * An instance of `element` has a list of elements plus a style. The style's interpretation depends on the renderer, but it can be one of a few special symbols (such as `'bold`) that are recognized by all renderers.
 - * An instance of `link-element` has a tag for the target of the link.
 - * An instance of `target-element` has a tag to be referenced by `link-elements`. An instance of the subtype `toc-target-element` is treated like a kind of section label, to be shown in the “on this page” table for HTML output.
 - * An instance of `index-element` has a tag (as a target), a list of strings for the keywords (for sorting and search), and a list of elements to appear in the end-of-document index.
 - * An instance of `collect-element` has a procedure that is called in the collect pass of document processing to record information used by later passes.
 - * A *part-relative element* is an instance of `part-relative-element`, which has a procedure that is called in the collect pass of document processing to obtain *content* (i.e., a list of *elements*). When the part-relative element's procedure is called, collected information is not yet available, but information about the enclosing parts is available.
 - * A *delayed element* is an instance of `delayed-element`, which has a procedure that is called in the resolve pass of document processing to obtain *content* (i.e., a list of *elements*).
 - * An instance of `aux-element` is excluded in the text of a link when it appears in a referenced section name.
 - * An instance of `hover-element` adds text to show in render HTML when the mouse hovers over the elements.
 - * An instance of `script-element` provides script code (usually Javascript) to run in the browser to generate the element; the element's normal content is used when scripting is disabled in the browser, or for rendering to other formats.
- A *delayed block* is an instance of `delayed-block`, which has a procedure that is called in the resolve pass of document processing to obtain a *block*.

4.2 Tags

A *tag* is a list containing a symbol and either a string, a `generated-tag` instance, or an arbitrary list. The symbol effectively identifies the type of the tag, such as `'part` for a tag that links to a part, or `'def` for a Scheme function definition. The symbol also effectively determines the interpretation of the second half of the tag.

A part can have a *tag prefix*, which is effectively added onto the second item within each tag whose first item is `'part` or `'tech`. The prefix is added to a string value by creating a list containing the prefix and string, and it is added to a list value using `cons`; a prefix is not added to a `generated-tag` instance. The prefix is used for reference outside the part, including the use of tags in the part's `tags` field. Typically, a document's main part has a tag prefix that applies to the whole document; references to sections and defined terms within the document from other documents must include, while references within the same document omit the prefix. Part prefixes can be used within a document as well, to help disambiguate references within the document.

Some procedures accept a “tag” that is just the string part of the full tag, where the symbol part is supplied automatically. For example, `section` and `secref` both accept a string “tag”, where `'part` is implicit.

4.3 Collected and Resolved Information

The collect pass, resolve pass, and render pass processing steps all produce information that is specific to a rendering mode. Concretely, the operations are all represented as methods on a `render%` object.

The result of the `collect` method is a `collect-info` instance. This result is provided back as an argument to the `resolve` method, which produces a `resolve-info` value that encapsulates the results from both iterations. The `resolve-info` value is provided back to the `resolve` method for final rendering.

Optionally, before the `resolve` method is called, serialized information from other documents can be folded into the `collect-info` instance via the `deserialize-info` method. Other methods provide serialized information out of the collected and resolved records.

During the collect pass, the procedure associated with a `collect-element` instance can register information with `collect-put!`.

During the resolve pass, collected information for a part can be extracted with `part-collected-info`, which includes a part's number and its parent part (or `#f`). More generally, the `resolve-get` method looks up information previously collected. This resolve-time information is normally obtained by the procedure associated with a delayed block or delayed element.

The `resolve-get` information accepts both a `part` and a `resolve-info` argument. The `part` argument enables searching for information in each enclosing part before sibling parts.

4.4 Structure Reference

```
(struct part (tag-prefix
             tags
             title-content
             style
             to-collect
             flow
             parts))
tag-prefix : (or/c false/c string?)
tags : (listof tag?)
title-content : (or/c false/c list?)
style : any/c
to-collect : list?
flow : flow?
parts : (listof part?)
```

The `tag-prefix` field determines the optional tag prefix for the part.

The `tags` indicates a list of tags that each link to the section.

The `title-content` field holds the part's title, if any.

The `style` field is normally either a symbol or a list of symbols. The currently recognized style symbols (alone or in a list) are as follows:

- `'toc` — sub-parts of the part are rendered on separate pages for multi-page HTML mode.
- `'index` — the part represents an index.
- `'reveal` — shows sub-parts when this part is displayed in a table-of-contents panel in HTML output (which normally shows only the top-level sections).
- `'hidden` — the part title is not shown in rendered output.
- `'no-toc` — as a style for the main part of a document, causes the HTML output to not include a margin box for the main table of contents; the “on this page” box that contains `toc-element` and `toc-target-element` links (and that only includes an “on this page” label for multi-page documents) takes on the location and color of the main table of contents, instead.

The `to-collect` field contains content that is inspected during the collect pass, but ignored in later passes (i.e., it doesn't directly contribute to the output).

The `flow` field contains the part's initial flow (before sub-parts).

The `parts` field contains sub-parts.

```
(struct (unnumbered-part part) ())
```

Although a section number is computed for an “unnumbered” section during the collect pass, the number is not rendered.

```
(struct (versioned-part part) (version))
  version : (or/c string? false/c)
```

Supplies a version number for this part and its sub-parts (except as overridden). A `#f` version is the same as not supplying a version.

The version number may be used when rendering a document. At a minimum, a version is rendered when it is attached to a part representing the whole document. The default version for a document is `(version)`.

```
(struct flow (paragraphs))
  paragraphs : (listof flow-element?)
```

A flow has a list of blocks.

```
(struct paragraph (content))
  content : list?
```

A paragraph has a list of elements.

```
(struct (styled-paragraph paragraph) (style))
  style : any/c
```

The `style` is normally a string that corresponds to a CSS class for HTML output.

```
(struct table (style flowss))
  style : any/c
  flowss : (listof (listof (or/c flow? (one-of/c 'cont))))
```

A table has, roughly, a list of list of flows. A cell in the table can span multiple columns by using `'cont` instead of a flow in the following columns (i.e., for all but the first in a set of cells that contain a single flow).

```
(struct itemization (flows))
```

```
flows : (listof flow?)
```

A itemization has a list of flows.

```
(struct blockquote (style paragraphs))
  style : any/c
  paragraphs : (listof flow-element?)
```

A blockquote has a style and a list of blocks. The `style` field is normally a string that corresponds to a CSS class for HTML output.

```
(struct delayed-block (resolve))
  resolve : (any/c part? resolve-info? . -> . flow-element?)
```

The `resolve` procedure is called during the resolve pass to obtain a normal block. The first argument to `resolve` is the renderer.

```
(struct element (style content))
  style : any/c
  content : list?
```

The `style` field is normally either

- a string, which corresponds to a CSS class for HTML output and a macro name for Latex output;
- one of the symbols that all renderers recognize: `'tt`, `'italic`, `'bold`, `'sf`, `'subscript`, `'superscript`, `'hspace`, or `'newline` (which renders a line break independent of the `content`);
- a list of the form `(list 'color name)` or `(list 'color byte byte byte)` to set the text color, where `name` is one of `"white"`, `"black"`, `"red"`, `"green"`, `"blue"`, `"cyan"`, `"magenta"`, or `"yellow"`, or three `bytes` specify RGB values;
- a list of the form `(list 'bg-color name)` or `(list 'bg-color byte byte byte)` to set the text background color (with the same constraints and meanings as for `'color`);
- an instance of `target-url` to generate a hyperlink;
- an instance of `image-file` to support an inline image; or
- an instance of `with-attributes`, which combines a base style with a set of additional HTML attributes.

The `content` field is a list of elements.

```
(struct (target-element element) (tag))
  tag : tag?
```

Declares the content as a hyperlink target for `tag`.

```
(struct (toc-target-element target-element) ())
```

Like `target-element`, the content is also a kind of section label to be shown in the “on this page” table for HTML output.

```
(struct (toc-element element) (toc-content))
  toc-content : list?
```

Similar to `toc-target-element`, but with specific content for the “on this page” table specified in the `toc-content` field.

```
(struct (link-element element) (tag))
  tag : tag?
```

Hyperlinks the content to `tag`.

```
(struct (index-element element) (tag plain-seq entry-seq desc))
  tag : tag?
  plain-seq : (and/c pair? (listof string?))
  entry-seq : list?
  desc : any/c
```

The `plain-seq` specifies the keys for sorting, where the first element is the main key, the second is a sub-key, etc. For example, an “night” portion of an index might have sub-entries for “night, things that go bump in” and “night, defender of the”. The former would be represented by `plain-seq '("night" "things that go bump in")`, and the latter by `'("night" "defender of the")`. Naturally, single-element `plain-seq` lists are the common case, and at least one word is required, but there is no limit to the word-list length. The strings in `plain-seq` must not contain a newline character.

The `entry-seq` list must have the same length as `plain-seq`. It provides the form of each key to render in the final document.

The `desc` field provides additional information about the index entry as supplied by the entry creator. For example, a reference to a procedure binding can be recognized when `desc` is

an instance of `procedure-index-desc`. See `scribble/manual-struct` for other typical types of `desc` values.

See also `index`.

```
(struct (aux-element element) ())
```

Instances of this structure type are intended for use in titles, where the auxiliary part of the title can be omitted in hyperlinks. See, for example, `secref`.

```
(struct (hover-element element) (text))
  text : string?
```

The `text` is displayed in HTML output when the mouse hovers over the element's content.

```
(struct (script-element element) (type script))
  type : string?
  script : (or/c path-string?
            (listof string?))
```

For HTML rendering, when scripting is enabled in the browser, `script` is used for the element instead of its normal content—it can be either path naming a script file to refer to, or the contents of the script. The `type` string is normally `"text/javascript"`.

```
(struct delayed-element (resolve sizer plain))
  resolve : (any/c part? resolve-info? . -> . list?)
  sizer : (-> any/c)
  plain : (-> any/c)
```

The `render` procedure's arguments are the same as for `delayed-block`, but the result is content (i.e., a list of elements). Unlike `delayed-block`, the result of the `render` procedure's argument is remembered on the first call for re-use for a particular resolve pass.

The `sizer` field is a procedure that produces a substitute element for the delayed element for the purposes of determining the delayed element's width (see `element-width`).

The `plain` field is a procedure that produces a substitute element when needed before the collect pass, such as when `element->string` is used before the collect pass.

```
(struct part-relative-element (resolve sizer plain))
  resolve : (collect-info? . -> . list?)
  sizer : (-> any/c)
  plain : (-> any/c)
```

Similar to `delayed-block`, but the replacement content is obtained in the collect pass by calling the function in the `resolve` field.

The `resolve` function can call `collect-info-parents` to obtain a list of parts that enclose the element, starting with the nearest enclosing section. Functions like `part-collected-info` and `collected-info-number` can extract information like the part number.

```
(struct (collect-element element) (collect))
  collect : (collect-info . -> . any)
```

Like `element`, but the `collect` procedure is called during the collect pass. The `collect` procedure normally calls `collect-put!`.

Unlike `delayed-element` or `part-relative-element`, the element remains intact (i.e., it is not replaced) by either the collect pass or resolve pass.

```
(struct with-attributes (style assoc))
  style : any/c
  assoc : (listof (cons/c symbol? string?))
```

Used for an `element`'s style to combine a base style with arbitrary HTML attributes.

```
(struct collected-info (number parent info))
  number : (listof (or/c false/c integer?))
  parent : (or/c false/c part?)
  info : any/c
```

Computed for each part by the collect pass.

```
(struct target-url (addr style))
  addr : path-string?
  style : any/c
```

Used as a style for an `element`. The `style` at this layer is a style for the hyperlink.

```
(struct image-file (path scale))
  path : (or/c path-string?
          (cons/c 'collects (listof bytes?)))
  scale : real?
```

Used as a style for an `element` to inline an image. The `path` field can be a result of `path->main-collects-relative`.

```
(block? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` is a `paragraph`, `table`, `itemization`, `blockquote`, or `delayed-block`, `#f` otherwise.

```
(tag? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` is acceptable as a link tag, which is a list containing a symbol and either a string, a `generated-tag` instance, or a list (of arbitrary values).

```
(struct generated-tag ())
```

A placeholder for a tag to be generated during the "collect" "\n" "pass". Use `tag-key` to convert a tag containing a `generated-tag` instance to one containing a string.

```
(content->string content) → string?  
  content : list?  
(content->string content renderer p info) → string?  
  content : list?  
  renderer : any/c  
  p : part?  
  info : resolve-info?
```

Converts a list of elements to a single string (essentially rendering the content as “plain text”).

If `p` and `info` arguments are not supplied, then a pre-“collect” substitute is obtained for delayed elements. Otherwise, the two arguments are used to force the delayed element (if it has not been forced already).

```
(element->string element) → string?  
  element : any/c  
(element->string element renderer p info) → string?  
  element : any/c  
  renderer : any/c  
  p : part?  
  info : resolve-info?
```

Like `content->string`, but for a single element.

```
(element-width element) → nonnegative-exact-integer?  
  element : any/c
```

Returns the width in characters of the given element.

```
(block-width e) → nonnegative-exact-integer?  
  e : block?
```

Returns the width in characters of the given block.

```
(struct collect-info (ht  
                    ext-ht  
                    parts  
                    tags  
                    gen-prefix  
                    relatives  
                    parents))  
  
ht : any/c  
ext-ht : any/c  
parts : any/c  
tags : any/c  
gen-prefix : any/c  
relatives : any/c  
parents : (listof part?)
```

Encapsulates information accumulated (or being accumulated) from the collect pass. The fields are exposed, but not currently intended for external use, except that `collect-info-parents` is intended for external use.

```
(struct resolve-info (ci delays undef))  
  ci : any/c  
  delays : any/c  
  undef : any/c
```

Encapsulates information accumulated (or being accumulated) from the resolve pass. The fields are exposed, but not currently intended for external use.

```
(info-key? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` is an *info key*: a list of at least two elements whose first element is a symbol. The result is `#f` otherwise.

For a list that is an info tag, the interpretation of the second element of the list is effectively determined by the leading symbol, which classifies the key. However, a `#f` value as the second element has an extra meaning: collected information mapped by such info keys is not propagated out of the part where it is collected; that is, the information is available within the part and its sub-parts, but not in ancestor or sibling parts.

Note that every tag is an info key.

```
(collect-put! ci key val) → void?  
  ci : collect-info?  
  key : info-key?  
  val : any/c
```

Registers information in `ci`. This procedure should be called only during the collect pass.

```
(resolve-get p ri key) → any/c  
  p : (or/c part? false/c)  
  ri : resolve-info?  
  key : info-key?
```

Extract information during the resolve pass or render pass for `p` from `ri`, where the information was previously registered during the collect pass. See also §4.3 “Collected and Resolved Information”.

The result is `#f` if the no value for the given key is found. Furthermore, the search failure is recorded for potential consistency reporting, such as when `setup-plt` is used to build documentation.

```
(resolve-get/ext? p ri key) → any/c boolean?  
  p : (or/c part? false/c)  
  ri : resolve-info?  
  key : info-key?
```

Like `render-get`, but returns a second value to indicate whether the resulting information originated from an external source (i.e., a different document).

```
(resolve-search dep-key p ri key) → void?  
  dep-key : any/c  
  p : (or/c part? false/c)  
  ri : resolve-info?  
  key : info-key?
```

Like `resolve-get`, but a shared `dep-key` groups multiple searches as a single request for

the purposes of consistency reporting and dependency tracking. That is, a single success for the same *dep-key* means that all of the failed attempts for the same *dep-key* have been satisfied. However, for dependency checking, such as when using `setup-plt` to rebuild documentation, all attempts are recorded (in case external changes mean that an earlier attempt would succeed next time).

```
(resolve-get/tentative p ri key) → any/c
  p : (or/c part? false/c)
  ri : resolve-info?
  key : info-key?
```

Like `resolve-search`, but without dependency tracking. For multi-document settings where dependencies are normally tracked, such as when using `setup-plt` to build documentation, this function is suitable for use only for information within a single document.

```
(resolve-get-keys p ri pred) → list?
  p : (or/c part? false/c)
  ri : resolve-info?
  pred : (info-key? . -> . any/c)
```

Applies *pred* to each key mapped for *p* in *ri*, returning a list of all keys for which *pred* returns a true value.

```
(part-collected-info p ri) → collected-info?
  p : part?
  ri : resolve-info?
```

Returns the information collected for *p* as recorded within *ri*.

```
(tag-key t ri) → tag?
  t : tag?
  ri : resolve-info?
```

Converts a `generated-tag` value with *t* to a string.

5 Renderer

A renderer is an object that provides two main methods: `collect` and `render`. The first method is called to collect global information about the document, including information that spans multiple documents rendered together; the collection pass tends to be format-independent, and it usually implemented completely by the base renderer. The latter method generates the actual output, which is naturally specific to a particular format.

5.1 Base Renderer

```
(require scribble/base-renderer)
```

The `scribble/base-renderer` module provides `render%`, which implements the core of a renderer. This rendering class must be refined with a mixin from `scribble/text-renderer`, `scribble/html-renderer`, or `scribble/latex-renderer`.

The mixin structure is meant to support document-specific extensions to the renderers. For example, the `scribble` command-line tool might, in the future, extract rendering mixins from a document module (in addition to the document proper).

See the "`base-renderer.ss`" source for more information about the methods of the renderer. Documents built with higher layers, such as `scribble/manual`, generally do not call the render object's methods directly.

```
render% : class?  
superclass: object%
```

Represents a renderer.

```
(new render%  
  [dest-dir dest-dir]  
  [[refer-to-existing-files refer-to-existing-files]  
  [root-path root-path]])  
→ (is-a?/c render%)  
dest-dir : path-string?  
refer-to-existing-files : any/c = #f  
root-path : (or/c path-string? false/c) = #f
```

Creates a renderer whose output will go to `dest-dir`. For example, `dest-dir` could name the directory containing the output Latex file, the HTML file for a single-file output, or the output sub-directory for multi-file HTML output.

If `root-path` is not `#f`, it is normally the same as `dest-dir` or a parent of `dest-dir`. It causes cross-reference information to record destination files rel-

ative to *root-path*; when cross-reference information is serialized, it can be deserialized via `deserialize-info` with a different root path (indicating that the destination files have moved).

```
(send a-render collect srcs dests) → collect-info?  
  srcs : (listof part?)  
  dests : (listof path-string?)
```

Performs the collect pass. See `render` for information on the *dests* argument.

```
(send a-render resolve srcs dests ci) → resolve-info?  
  srcs : (listof part?)  
  dests : (listof path-string?)  
  ci : collect-info?
```

Performs the resolve pass. See `render` for information on the *dests* argument.

```
(send a-render render srcs dests ri) → void?  
  srcs : (listof part?)  
  dests : (listof path-string?)  
  ri : resolve-info?
```

Produces the final output.

The *dests* provide names of files for Latex or single-file HTML output, or names of sub-directories for multi-file HTML output. If the *dests* are relative, they're relative to the current directory; normally, they should indicate a path within the *dest-dir* supplied on initialization of the `render%` object.

```
(send a-render serialize-info ri) → any/c  
  ri : resolve-info?
```

Serializes the collected info in *ri*.

```
(send a-render deserialize-info v  
                                     ci  
                                     [#:root root-path]) → void?  
  v : any/c  
  ci : collect-info?  
  root-path : (or/c path-string? false/c) = #f
```

Adds the deserialized form of *v* to *ci*.

If *root-path* is not `#f`, then file paths that are recorded in *ci* as relative to an instantiation-supplied *root-path* are deserialized as relative instead to the given *root-path*.

5.2 Text Renderer

```
(require scribble/text-render)
```

```
render-mixin : (class? . -> . class?)  
argument extends/implements: render%
```

Specializes a `render%` class for generating plain text.

5.3 HTML Renderer

```
(require scribble/html-render)
```

```
render-mixin : (class? . -> . class?)  
argument extends/implements: render%
```

Specializes a `render%` class for generating HTML output.

```
(send a-render set-external-tag-path url) → void?  
url : string?
```

Configures the renderer to redirect links to external via `url`, adding a tag query element to the end of the URL that contains the Base64-encoded, `printed`, serialized original tag (in the sense of `link-element`) for the link.

```
render-multi-mixin : (class? . -> . class?)  
argument extends/implements: render%
```

Further specializes a rendering class produced by `render-mixin` for generating multiple HTML files.

5.4 Latex Renderer

```
(require scribble/latex-render)
```

```
render-mixin : (class? . -> . class?)
```

argument extends/implements: [render%](#)

Specializes a [render%](#) class for generating Latex input.

6 Decoding Text

```
(require scribble/decode)
```

The `scribble/decode` library helps you write document content in a natural way—more like plain text, except for `@` escapes. Roughly, it processes a stream of strings to produce instances of the `scribble/struct` datatypes (see §4 “Structures And Processing”).

At the flow level, decoding recognizes a blank line as a paragraph separator. At the paragraph-content level, decoding makes just a few special text conversions:

- `---`: converted to `'mdash`, which the HTML render outputs as an en-dash surrounded by space (so don't put spaces around `---` in a document)
- `--`: converted to `'ndash`
- `'‘`: converted to `'ldquo`, which is fancy open quotes: “
- `'’`: converted to `'rdquo`, which is fancy closing quotes: ”
- `'``: converted to `'rsquo`, which is a fancy apostrophe: ’

Some functions `decode` a sequence of `pre-flow` or `pre-content` arguments using `decode-flow` or `decode-content`, respectively. For example, the `bold` function accepts any number of `pre-content` arguments, so that in

```
@bold{‘‘apple’’}
```

the `‘‘apple’’` argument is decoded to use fancy quotes, and then it is bolded.

```
(decode lst) → part?  
lst : list?
```

Decodes a document, producing a part. In `lst`, instances of `splICE` are inlined into the list. An instance of `title-decl` supplies the title for the part, plus tag, style and version information. Instances of `part-index-decl` (that precede any sub-part) add index entries that point to the section. Instances of `part-collect-decl` add elements to the part that are used only during the collect pass. Instances of `part-tag-decl` add hyperlink tags to the section title. Instances of `part-start` at level 0 trigger sub-part parsing. Instances of `section` trigger are used as-is as subsections, and instances of `paragraph` and other flow-element datatypes are used as-is in the enclosing flow.

```
(decode-part lst tags title depth) → part?  
lst : list?
```

```
tags : (listof string?)
title : (or/c false/c list?)
depth : excat-nonnegative-integer?
```

Like `decode`, but given a list of tag string for the part, a title (if `#f`, then a `title-decl` instance is used if found), and a depth for `part-starts` to trigger sub-part parsing.

```
(decode-flow lst) → (listof flow-element?)
lst : list?
```

Decodes a flow. A sequence of two or more newlines separated only by whitespace counts is parsed as a paragraph separator. In `lst`, instances of `splice` are inlined into the list. Instances of `paragraph` and other flow-element datatypes are used as-is in the enclosing flow.

```
(decode-paragraph lst) → paragraph?
lst : list?
```

Decodes a paragraph.

```
(decode-content lst) → list?
lst : list?
```

Decodes a sequence of elements.

```
(decode-elements lst) → list?
lst : list?
```

An alias for `decode-content`.

```
(decode-string s) → list?
s : string?
```

Decodes a single string to produce a list of elements.

```
(whitespace? s) → boolean?
s : string?
```

Returns `#t` if `s` contains only whitespace, `#f` otherwise.

```
(struct title-decl (tag-prefix tags version style content))
```

```
tag-prefix : (or/c false/c string?)
tags : (listof string?)
version : (or/c string? false/c)
style : any/c
content : list?
```

See [decode](#) and [decode-part](#). The `tag-prefix` and `style` fields are propagated to the resulting `part`.

```
(struct part-start (depth tag-prefix tags style title))
depth : integer?
tag-prefix : (or/c false/c string?)
tags : (listof string?)
style : any/c
title : list?
```

Like [title-decl](#), but for a sub-part. See [decode](#) and [decode-part](#).

```
(struct part-index-decl (plain-seq entry-seq))
plain-seq : (listof string?)
entry-seq : list?
```

See [decode](#). The two fields are as for [index-element](#).

```
(struct part-collect-decl (element))
element : element?
```

See [decode](#).

```
(struct part-tag-decl (tag))
tag : tag?
```

See [decode](#).

```
(struct splice (run))
run : list?
```

See [decode](#), [decode-part](#), and [decode-flow](#).

```
(clean-up-index-string str) → string?
str : string?
```

Trims leading and trailing whitespace, and converts non-empty sequences of whitespace to a single space character.

7 Document Language

```
#lang scribble/doclang
```

The `scribble/doclang` language provides everything from `scheme/base`, except that it replaces the `#!/module-begin` form.

The `scribble/doclang` `#!/module-begin` essentially packages the body of the module into a call to `decode`, binds the result to `doc`, and exports `doc`.

Any module-level form other than an expression (e.g., a `require` or `define`) remains at the top level, and the `doc` binding is put at the end of the module. As usual, a module-top-level `begin` slices into the module top level.

8 Document Reader

```
#lang scribble/doc
```

The `scribble/doc` language is the same as `scribble/doclang`, except that `read-syntax-inside` is used to read the body of the module. In other words, the module body starts in Scribble “text” mode instead of S-expression mode.

9 Basic Document Forms

```
(require scribble/basic)
```

The `scribble/basic` library provides functions and forms that can be used from code written either in Scheme or with `@` expressions.

For example, the `title` and `italic` functions might be called from Scheme as

```
(title #:tag "how-to"
       "How to Design " (italic "Great") " Programs")
```

or with an `@` expression as

```
@title[#:tag "how-to"]{How to Design @italic{Great} Programs}
```

Although the procedures are mostly design to be used from `@` mode, they are easier to document in Scheme mode (partly because we have `scribble/manual`).

9.1 Document Structure

```
(title [#:tag tag
        #:tag-prefix tag-prefix
        #:style style
        #:version vers]
       pre-content ...+) → title-decl?
tag : (or/c false/c string?) = #f
tag-prefix : (or/c false/c string? module-path?) = #f
style : any/c = #f
vers : (or/c string? false/c) = #f
pre-content : any/c
```

Generates a `title-decl` to be picked up by `decode` or `decode-part`. The decoded `pre-content` (i.e., parsed with `decode-content`) supplies the title content. If `tag` is `#f`, a tag string is generated automatically from the content. The tag string is combined with the symbol `'part` to form the full tag.

A style of `'toc` causes sub-sections to be generated as separate pages in multi-page HTML output. A style of `'index` indicates an index section whose body is rendered in two columns for Latex output.

The `tag-prefix` argument is propagated to the generated structure (see §4.2 “Tags”). If `tag-prefix` is a module path, it is converted to a string using `module-path-prefix->string`.

The `vers` argument is propagated to the `title-decl` structure.

The section title is automatically indexed by `decode-part`. For the index key, leading whitespace and a leading “A”, “An”, or “The” (followed by more whitespace) is removed.

```
(section [#:tag tag
         #:tag-prefix tag-prefix
         #:style style]
        pre-content ...+) → part-start?
tag : (or/c false/c string?) = #f
tag-prefix : (or/c false/c string? module-path?) = #f
style : any/c = #f
pre-content : any/c
```

Like `title`, but generates a `part-start` of depth 0 to be by `decode` or `decode-part`.

```
(subsection [#:tag tag
            #:tag-prefix tag-prefix
            #:style style]
           pre-content ...+) → part-start?
tag : (or/c false/c string?) = #f
tag-prefix : (or/c false/c string? module-path?) = #f
style : any/c = #f
pre-content : any/c
```

Like `section`, but generates a `part-start` of depth 1.

```
(subsubsection [#:tag tag
               #:tag-prefix tag-prefix
               #:style style]
              pre-content ...+) → part-start?
tag : (or/c false/c string?) = #f
tag-prefix : (or/c false/c string? module-path?) = #f
style : any/c = #f
pre-content : any/c
```

Like `section`, but generates a `part-start` of depth 2.

```
(subsubsub*section [#:tag tag
                   #:tag-prefix tag-prefix
                   #:style style]
                  pre-content ...+) → paragraph?
tag : (or/c false/c string?) = #f
```

```
tag-prefix : (or/c false/c string? module-path?) = #f
style : any/c = #f
pre-content : any/c
```

Similar to `section`, but merely generates a paragraph that looks like an unnumbered section heading (for when the nesting gets too deep to include in a table of contents).

```
(itemize itm ...) → itemization?
itm : (or/c whitespace? an-item?)
```

Constructs an itemization given a sequence of items constructed by `item`. Whitespace strings among the `itms` are ignored.

```
(item pre-flow ...) → item?
pre-flow : any/c
```

Creates an item for use with `itemize`. The decoded `pre-flow` (i.e., parsed with `decode-flow`) is the item content.

```
(item? v) → boolean?
v : any/c
```

Returns `#t` if `v` is an item produced by `item`, `#f` otherwise.

```
(include-section module-path)
```

Requires `module-path` and returns its `doc` export (without making any imports visible to the enclosing context). Since this form expands to `require`, it must be used in a module or top-level context.

```
(module-path-prefix->string mod-path) → string?
mod-path : module-path?
```

Converts a module path to a string by resolving it to a path, and using `path->main-collects-relative`.

9.2 Text Styles

```
(elem pre-content ...) → element?
pre-content : any/c
```

Wraps the decoded *pre-content* as an element with style *#f*.

```
(aux-elem pre-content ...) → element?  
pre-content : any/c
```

Like *elem*, but creates an *aux-element*.

```
(italic pre-content ...) → element?  
pre-content : any/c
```

Like *elem*, but with style *'italic*

```
(bold pre-content ...) → element?  
pre-content : any/c
```

Like *elem*, but with style *'bold*

```
(tt pre-content ...) → element?  
pre-content : any/c
```

Like *elem*, but with style *'tt*

```
(subscript pre-content ...) → element?  
pre-content : any/c
```

Like *elem*, but with style *'subscript*

```
(superscript pre-content ...) → element?  
pre-content : any/c
```

Like *elem*, but with style *'superscript*

```
(hspace n) → element?  
n : nonnegative-exact-integer?
```

Produces an element containing *n* spaces and style *'hspace*.

```
(span-class style-name pre-content ...) → element?  
style-name : string?  
pre-content : any/c
```

Wraps the decoded *pre-content* as an element with style *style-name*.

9.3 Indexing

```
(index words pre-content ...) → index-element?  
  words : (or/c string? (listof string?))  
  pre-content : any/c
```

Creates an index element given a plain-text string—or list of strings for a hierarchy, such as `'("strings" "plain")` for a “plain” entry below a more general “strings” entry. As index keys, the strings are “cleaned” using `clean-up-index-strings`. The strings (without clean-up) also serve as the text to render in the index. The decoded *pre-content* is the text to appear inline as the index target.

```
(index* words word-contents pre-content ...) → index-element?  
  words : (listof string?)  
  word-contents : (listof list?)  
  pre-content : any/c
```

Like `index`, except that *words* must be a list, and the list of contents render in the index (in parallel to *words*) is supplied as *word-contents*.

```
(as-index pre-content ...) → index-element?  
  pre-content : any/c
```

Like `index`, but the word to index is determined by applying `content->string` on the decoded *pre-content*.

```
(section-index word ...) → part-index-decl?  
  word : string?
```

Creates a `part-index-decl` to be associated with the enclosing section by `decode`. The *words* serve as both the keys and as the rendered forms of the keys.

```
(index-section [#:tag tag]) → part?  
  tag : (or/c false/c string?) = "doc-index"
```

Produces a part that shows the index the enclosing document. The optional *tag* argument is used as the index section’s tag.

9.4 Tables of Contents

`(table-of-contents)` → delayed-flow-element?

Returns a delayed flow element that expands to a table of contents for the enclosing section. For LaTeX output, however, the table of contents currently spans the entire enclosing document.

`(local-table-of-contents)` → delayed-flow-element?

Returns a delayed flow element that may expand to a table of contents for the enclosing section, depending on the output type. For multi-page HTML output, the flow element is a table of contents; for Latex output, the flow element is empty.

10 Scheme

```
(require scribble/scheme)
```

The `scribble/scheme` library provides utilities for typesetting Scheme code. The `scribble/manual` forms provide a higher-level interface.

To do: document this library!

11 Manual Forms

```
(require scribble/manual)
```

The `scribble/manual` library provides all of `scribble/basic`, plus additional functions that are relatively specific to writing PLT Scheme documentation.

11.1 Typesetting Code

```
(schemeblock datum ...)
```

Typesets the `datum` sequence as a table of Scheme code inset by two spaces. The source locations of the `datums` determine the generated layout. For example,

```
(schemeblock
  (define (loop x)
    (loop (not x))))
```

produces the output

```
(define (loop x)
  (loop (not x)))
```

with the `(loop (not x))` indented under `define`, because that's the way it is indented the use of `schemeblock`.

Furthermore, `define` is typeset as a keyword (bold and black) and as a hyperlink to `define`'s definition in the reference manual, because this document was built using a for-label binding of `define` (in the source) that matches a definition in the reference manual. Similarly, `not` is a hyperlink to its definition in the reference manual.

Use `unsyntax` to escape back to an expression that produces an `element`. For example,

```
(schemeblock
  (+ 1 #,(elem (scheme x) (subscript "2"))))
```

produces

```
(+ 1  $x_2$ )
```

The `unsyntax` form is recognized via `free-identifier=?`, so if you want to typeset code that includes `unsyntax`, you can simply hide the usual binding:

```
(schemeblock
  (let ([unsyntax #f])
```

```
(schemeblock
  #'(+ 1 #,x)))
```

Or use SCHEMEBLOCK, whose escape form is UNSYNTAX instead of unsyntax.

A few other escapes are recognized symbolically:

- `(code:line datum ...)` typesets as the sequence of *datums* (i.e., without the `code:line` wrapper).
- `(code:comment datum)` typesets like *datum*, but colored as a comment and prefixed with a semi-colon. A typical *datum* escapes from Scheme-typesetting mode using `unsyntax` and produces a paragraph using `t`:

```
(code:comment #, @t{this is a comment})
```

- `code:blank` typesets as a blank space.
- `_id` typesets as *id*, but colored as a variable (like `schemevarfont`); this escape applies only if `_id` has no for-label binding and is not specifically colored as a subform non-terminal via `deform`, a variable via `defproc`, etc.

```
(SCHEMEBLOCK datum ...)
```

Like `schemeblock`, but with the expression escape `UNSYNTAX` instead of `unsyntax`.

```
(schemeblock0 datum ...)
```

Like `schemeblock`, but without insetting the code.

```
(SCHEMEBLOCK0 datum ...)
```

Like `SCHEMEBLOCK`, but without insetting the code.

```
(schemeinput datum ...)
```

Like `schemeblock`, but the *datum* are typeset after a prompt representing a REPL.

```
(schememod lang datum ...)
```

Like `schemeblock`, but the *datum* are typeset inside a `#lang`-form module whose language is *lang*.

`(scheme datum ...)`

Like `schemeblock`, but typeset on a single line and wrapped with its enclosing paragraph, independent of the formatting of `datum`.

`(SCHEME datum ...)`

Like `scheme`, but with the `UNSYNTAX` escape like `schemeblock`.

`(schemeresult datum ...)`

Like `scheme`, but typeset as a REPL value (i.e., a single color with no hyperlinks).

`(schemeid datum ...)`

Like `scheme`, but typeset as an unbound identifier (i.e., no coloring or hyperlinks).

`(schememodname datum)`

Like `scheme`, but typeset as a module path. If `datum` is an identifier, then it is hyperlinked to the module path's definition as created by `defmodule`.

`(litchar str) → element?`
`str : string?`

Typesets `str` as a representation of literal text. Use this when you have to talk about the individual characters in a stream of text, as when documenting a reader extension.

`(verbatim [#:indent indent] str ...) → flow-element?`
`indent : integer? = 0`
`str : string?`

Typesets `str` as a table/paragraph in typewriter font with the linebreaks specified by newline characters in `str`. “Here strings” are often useful with `verbatim`.

`(schemefont pre-content ...) → element?`
`pre-content : any/c`

Typesets decoded `pre-content` as uncolored, unhyperlinked Scheme. This procedure is

useful for typesetting things like `#lang`, which are not [readable](#) by themselves.

```
(schemevalfont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as a value.

```
(schemeresultfont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as a REPL result.

```
(schemeidfont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as an identifier.

```
(schemevarfont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as a variable (i.e., an argument or sub-form in a procedure being documented).

```
(schemekeywordfont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as a syntactic form name.

```
(schemeparenfont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored like parentheses.

```
(schememetafont pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as meta-syntax, such as backquote or unquote.

```
(schemeerror pre-content ...) → element?  
pre-content : any/c
```

Like `schemefont`, but colored as error-message text.

```
(procedure pre-content ...) → element?  
pre-content : any/c
```

Typesets decoded `pre-content` as a procedure name in a REPL result (e.g., in typewriter font with a `#<procedure:` prefix and `>` suffix.).

```
(var datum)
```

Typesets `datum` as an identifier that is an argument or sub-form in a procedure being documented. Normally, the `defproc` and `defform` arrange for `scheme` to format such identifiers automatically in the description of the procedure, but use `var` if that cannot work for some reason.

```
(svar datum)
```

Like `var`, but for subform non-terminals in a form definition.

11.2 Documenting Modules

```
(defmodule id maybe-sources pre-flow ...)
```

```
maybe-sources =  
  | #:use-sources (mod-path ...)
```

Produces a sequence of flow elements (encaptured in a `splice`) to start the documentation for a module that can be required using the path `id`. The decoded `pre-flows` introduce the module, but need not include all of the module content.

Besides generating text, this form expands to a use of `declare-exporting` with `id`; the `#:use-sources` clause, if provided, is propagated to `declare-exporting`. Consequently, `defmodule` should be used at most once in a section, though it can be shadowed with `defmodules` in sub-sections.

Hyperlinks created by `schememodname` are associated with the enclosing section, rather than the local `id` text.

```
(defmodulelang id maybe-sources pre-flow ...)
```

Like `defmodule`, but documents `id` as a module path suitable for use by either `require` or `#lang`.

```
(defmodule* (id ...) maybe-sources pre-flow ...)
```

Like `defmodule`, but introduces multiple module paths instead of just one.

```
(defmodulelang* (id ...) maybe-sources pre-flow ...)
```

Like `defmodulelang`, but introduces multiple module paths instead of just one.

```
(defmodule*/no-declare (id ...) pre-flow ...)
```

Like `defmodule*`, but without expanding to `declare-exporting`. Use this form when you want to provide a more specific list of modules (e.g., to name both a specific module and one that combines several modules) via your own `declare-exporting` declaration.

```
(defmodulelang*/no-declare (id ...) pre-flow ...)
```

Like `defmodulelang*`, but without expanding to `declare-exporting`.

```
(declare-exporting mod-path ... maybe-sources)
```

```
maybe-sources = #:use-sources  
                | (mod-path ...)
```

Associates the `mod-paths` to all bindings defined within the enclosing section, except as overridden by other `declare-exporting` declarations in nested sub-sections. The list of `mod-paths` is shown, for example, when the user hovers the mouse over one of the bindings defined within the section.

More significantly, the first `mod-path` plus the `#:use-sources mod-paths` determine the binding that is documented by each `defform`, `defproc`, or similar form within the section that contains the `declare-exporting` declaration:

- If no `#:use-sources` clause is supplied, then the documentation applies to the given name as exported by the first `mod-path`.
- If `#:use-sources mod-paths` are supplied, then they are tried in order. The first one to provide an export with the same symbolic name and `free-label-identifier=?` to the given name is used as the documented binding. This binding is assumed to be the

same as the identifier as exported by the first *mod-path* in the `declare-exporting` declaration.

The initial *mod-paths* sequence can be empty if *mod-paths* are given with `#:use-sources`. In that case, the rendered documentation never reports an exporting module for identifiers that are documented within the section, but the *mod-paths* in `#:use-sources` provide a binding context for connecting (via hyperlinks) definitions and uses of identifiers.

The `declare-exporting` form should be used no more than once per section, since the declaration applies to the entire section, although overriding `declare-exporting` forms can appear in sub-sections.

11.3 Documenting Forms, Functions, Structure Types, and Values

```
(defproc prototype
  result-contract-expr-datum
  pre-flow ...)
```

prototype = (*id* *arg-spec* ...)
 | (*prototype* *arg-spec* ...)

arg-spec = (*arg-id* *contract-expr-datum*)
 | (*arg-id* *contract-expr-datum* *default-expr*)
 | (*keyword* *arg-id* *contract-expr-datum*)
 | (*keyword* *arg-id* *contract-expr-datum* *default-expr*)
 | *ellipses*
 | *ellipses+*

ellipses = ...

ellipses+ = ...+

Produces a sequence of flow elements (encapsulated in a `splice`) to document a procedure named *id*. Nesting *prototypes* corresponds to a curried function, as in `define`. The *id* is indexed, and it also registered so that `scheme-typeset` uses of the identifier (with the same `for-label` binding) are hyperlinked to this documentation.

A `defmodule` or `declare-exporting` form (or one of the variants) in an enclosing section determines the *id* binding that is being defined. The *id* should also have a `for-label` binding (as introduced by `(require (for-label ...))`) that matches the definition binding; otherwise, the defined *id* will not typeset correctly within the definition.

Each *arg-spec* must have one of the following forms:

(arg-id contract-expr-datum)

An argument whose contract is specified by *contract-expr-datum* which is typeset via `schemeblock0`.

(arg-id contract-expr-datum default-expr)

Like the previous case, but with a default value. All arguments with a default value must be grouped together, but they can be in the middle of required arguments.

(keyword arg-id contract-expr-datum)

Like the first case, but for a keyword-based argument.

(keyword arg-id contract-expr-datum default-expr)

Like the previous case, but with a default value.

...

Any number of the preceding argument. This form is normally used at the end, but keyword-based arguments can sensibly appear afterward. See also the documentation for `append` for a use of ... before the last argument.

...+

One or more of the preceding argument (normally at the end, like ...).

The *result-contract-expr-datum* is typeset via `schemeblock0`, and it represents a contract on the procedure's result.

The decoded *pre-flow* documents the procedure. In this description, references to *arg-ids* using `scheme`, `schemeblock`, etc. are typeset as procedure arguments.

The typesetting of all information before the *pre-flows* ignores the source layout, except that the local formatting is preserved for contracts and default-values expressions.

```
(defproc* ([prototype
           result-contract-expr-datum] ...)
          pre-flow ...)
```

Like `defproc`, but for multiple cases with the same `id`.

When an `id` has multiple calling cases, they must be defined with a single `defproc*`, so that a single definition point exists for the `id`. However, multiple distinct `ids` can also be defined by a single `defproc*`, for the case that it's best to document a related group of procedures at once.

```
(deform maybe-id maybe-literals form-datum pre-flow ...)
```

```
    maybe-id =
        | #:id id
```

```
maybe-literals =
        | #:literals (literal-id ...)
```

Produces a sequence of flow elements (encapsulated in a `splice`) to document a syntactic form named by `id` whose syntax described by `form-datum`. If no `#:id` is used to specify `id`, then `form-datum` must have the form `(id . datum)`.

The `id` is indexed, and it is also registered so that `scheme-typeset` uses of the identifier (with the same for-label binding) are hyperlinked to this documentation.

The `defmodule` or `declare-exporting` requirements, as well as the binding requirements for `id`, are the same as for `defproc`.

The decoded `pre-flow` documents the form. In this description, a reference to any identifier in `form-datum` via `scheme`, `schemeblock`, etc. is typeset as a sub-form non-terminal. If `#:literals` clause is provided, however, instances of the `literal-ids` are typeset normally (i.e., as determined by the enclosing context).

The typesetting of `form-datum` preserves the source layout, like `schemeblock`.

```
(deform* maybe-id maybe-literals [form-datum ...+] pre-flow ...)
```

Like `deform`, but for multiple forms using the same `id`.

```
(deform/subs maybe-id maybe-literals form-datum
             ([nonterm-id clause-datum ...+] ...)
             pre-flow ...)
```

Like `defform`, but including an auxiliary grammar of non-terminals shown with the `id` form. Each `nonterm-id` is specified as being any of the corresponding `clause-datums`, where the formatting of each `clause-datum` is preserved.

```
(defform*/subs maybe-id maybe-literals [form-datum ...]
  pre-flow ...)
```

Like `defform/subs`, but for multiple forms for `id`.

```
(defform/none maybe-literal form-datum pre-flow ...)
```

Like `defform`, but without registering a definition.

```
(defidform id pre-flow ...)
```

Like `defform`, but with a plain `id` as the form.

```
(specform maybe-literals datum pre-flow ...)
```

Like `defform`, but without indexing or registering a definition, and with indenting on the left for both the specification and the `pre-flows`.

```
(specsubform maybe-literals datum pre-flow ...)
```

Similar to `defform`, but without any specific identifier being defined, and the table and flow are typeset indented. This form is intended for use when refining the syntax of a non-terminal used in a `defform` or other `specsubform`. For example, it is used in the documentation for `defproc` in the itemization of possible shapes for `arg-spec`.

The `pre-flows` list is parsed as a flow that documents the procedure. In this description, a reference to any identifier in `datum` is typeset as a sub-form non-terminal.

```
(specsubform/subs maybe-literals datum
  ([nonterm-id clause-datum ...+] ...)
  pre-flow ...)
```

Like `specsubform`, but with a grammar like `defform/subs`.

```
(specspecsubform maybe-literals datum pre-flow ...)
```

Like `specsubform`, but indented an extra level. Since using `specsubform` within the body of `specsubform` already nests indentation, `specspecsubform` is for extra indentation without nesting a description.

```
(specspecsubform/subs maybe-literals datum
  ([nonterm-id clause-datum ...+] ...)
  pre-flow ...)
```

Like `specspecsubform`, but with a grammar like `defform/subs`.

```
(defparam id arg-id contract-expr-datum pre-flow ...)
```

Like `defproc`, but for a parameter. The `contract-expr-datum` serves as both the result contract on the parameter and the contract on values supplied for the parameter. The `arg-id` refers to the parameter argument in the latter case.

```
(defboolparam id arg-id pre-flow ...)
```

Like `defparam`, but the contract on a parameter argument is `any/c`, and the contract on the parameter result is `boolean?`.

```
(defthing id contract-expr-datum pre-flow ...)
```

Like `defproc`, but for a non-procedure binding.

```
(defstruct struct-name ([field-name contract-expr-datum] ...)
  flag-keywords
  pre-flow ...)
```

```
  struct-name = id
                | (id super-id)

  flag-keywords =
                | #:mutable
                | #:inspector #f
                | #:mutable #:inspector #f
```

Similar to `defform` or `defproc`, but for a structure definition.

```
(deftogether [def-expr ...] pre-flow ...)
```

Combines the definitions created by the *def-exprs* into a single definition box. Each *def-expr* should produce a definition point via *defproc*, *defform*, etc. Each *def-expr* should have an empty *pre-flow*; the decoded *pre-flow* sequence for the *deftogether* form documents the collected bindings.

```
(schemegrammar maybe-literals id clause-datum ...+)
```

```
maybe-literals =  
  | #:literals (literal-id ...)
```

Creates a table to define the grammar of *id*. Each identifier mentioned in a *clause-datum* is typeset as a non-terminal, except for the identifiers listed as *literal-ids*, which are typeset as with *scheme*.

```
(schemegrammar* maybe-literals [id clause-datum ...+] ...)
```

Like *schemegrammar*, but for typesetting multiple productions at once, aligned around the `=` and `||`.

11.4 Documenting Classes and Interfaces

```
(defclass id super (intf-id ...) pre-flow ...)
```

```
super = super-id  
  | (mixin-id super)
```

Creates documentation for a class *id* that is a subclass of *super* and implements each interface *intf-id*. Each identifier in *super* (except *object%*) and *intf-id* must be documented somewhere via *defclass* or *definterface*.

The decoding of the *pre-flow* sequence should start with general documentation about the class, followed by constructor definition (see *defconstructor*), and then field and method definitions (see *defmethod*). In rendered form, the constructor and method specification are indented to visually group them under the class definition.

```
(defclass/title id super (intf-id ...) pre-flow ...)
```

Like *defclass*, also includes a *title* declaration with the style `'hidden`. In addition, the constructor and methods are not left-indented.

This form is normally used to create a section to be rendered on its own HTML. The `'hid-`

den style is used because the definition box serves as a title.

```
(definterface id (intf-id ...) pre-flow ...)
```

Like `defclass`, but for an interfaces. Naturally, *pre-flow* should not generate a constructor declaration.

```
(definterface/title id (intf-id ...) pre-flow ...)
```

Like `definterface`, but for single-page rendering as in `defclass/title`.

```
(defmixin id (domain-id ...) (range-id ...) pre-flow ...)
```

Like `defclass`, but for a mixin. Any number of *domain-id* classes and interfaces are specified for the mixin's input requires, and any number of result classes and (more likely) interfaces are specified for the *range-id*. The *domain-ids* supply inherited methods.

```
(defmixin/title id (domain-id ...) (range-id ...) pre-flow ...)
```

Like `defmixin`, but for single-page rendering as in `defclass/title`.

```
(defconstructor (arg-spec ...) pre-flow ...)
```

```
arg-spec = (arg-id contract-expr-datum)  
          | (arg-id contract-expr-datum default-expr)
```

Like `defproc`, but for a constructor declaration in the body of `defclass`, so no return contract is specified. Also, the `new-style` keyword for each *arg-spec* is implicit from the *arg-id*.

```
(defconstructor/make (arg-spec ...) pre-flow ...)
```

Like `defconstructor`, but specifying by-position initialization arguments (for use with `make-object`) instead of by-name arguments (for use with `new`).

```
(defconstructor*/make [(arg-spec ...) ...] pre-flow ...)
```

Like `defconstructor/make`, but with multiple constructor patterns analogous `defproc*`.

```
(defconstructor/auto-super [(arg-spec ...) ...] pre-flow ...)
```

Like `defconstructor`, but the constructor is annotated to indicate that additional initialization arguments are accepted and propagated to the superclass.

```
(defmethod (id arg-spec ...)
  result-contract-expr-datum
  pre-flow ...)
```

Like `defproc`, but for a method within a `defclass` or `definterface` body.

```
(defmethod* ([ (id arg-spec ...)
  result-contract-expr-datum ] ...)
  pre-flow ...)
```

Like `defproc*`, but for a method within a `defclass` or `definterface` body.

```
(method class/intf-id method-id)
```

Creates a hyperlink to the method named by `method-id` in the class or interface named by `class/intf-id`. The hyperlink names the method, only; see also `xmethod`.

For-label binding information is used with `class/intf-id`, but not `method-id`.

```
(xmethod class/intf-id method-id)
```

Like `method`, but the hyperlink shows both the method name and the containing class/interface.

11.5 Documenting Signatures

```
(defsignature id (super-id ...) pre-flow ...)
```

Defines a signature `id` that extends the `super-id` signatures. Any elements defined in decoded `pre-flows`—including forms, procedures, structure types, classes, interfaces, and mixins—are defined as members of the signature instead of direct bindings. These definitions can be referenced through `sigelem` instead of `scheme`.

The decoded `pre-flows` inset under the signature declaration in the typeset output, so no new sections, etc. can be started.

```
(defsignature/splice id (super-id ...) pre-flow ...)
```

Like `defsignature`, but the decoded *pre-flows* are not typeset under the signature declaration, and new sections, etc. can be started in the *pre-flows*.

```
(signature-desc pre-flow ...) → any/c
  pre-flow : any/c
```

Produces an opaque value that `defsignature` recognizes to outdent in the typeset form. This is useful for text describing the signature as a whole to appear right after the signature declaration.

```
(sigelem sig-id id)
```

Typesets the identifier *id* with a hyperlink to its definition as a member of the signature named by *sig-id*.

11.6 Various String Forms

```
(emph pre-content ...) → element?
  pre-content : any/c
```

Typesets the decoded *pre-content* with emphasis (e.g., in italic).

```
(defterm pre-content ...) → element?
  pre-content : any/c
```

Typesets the decoded *pre-content* as a defined term (e.g., in italic). Consider using `deftech` instead, though, so that uses of `tech` can hyper-link to the definition.

```
(onscreen pre-content ...) → element?
  pre-content : any/c
```

Typesets the decoded *pre-content* as a string that appears in a GUI, such as the name of a button.

```
(menuitem menu-name item-name) → element?
  menu-name : string?
  item-name : string?
```

Typesets the given combination of a GUI's menu and item name.

`(filepath pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the decoded `pre-content` as a file name (e.g., in typewriter font and in quotes).

`(exec pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the decoded `pre-content` as a command line (e.g., in typewriter font).

`(envvar pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the given decoded `pre-content` as an environment variable (e.g., in typewriter font).

`(Flag pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the given decoded `pre-content` as a flag (e.g., in typewriter font with a leading `=`).

`(DFlag pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the given decoded `pre-content` a long flag (e.g., in typewriter font with two leading `=s`).

`(PFlag pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the given decoded `pre-content` as a `+` flag (e.g., in typewriter font with a leading `+`).

`(DPFlag pre-content ...)` → `element?`
`pre-content` : `any/c`

Typesets the given decoded `pre-content` a long `+` flag (e.g., in typewriter font with two leading `+`s).

```
(math pre-content ...) → element?
  pre-content : any/c
```

The decoded *pre-content* is further transformed:

- Any immediate `'rsquo` is converted to `'prime`.
- Parentheses and sequences of decimal digits in immediate strings are left as-is, but any other immediate string is italicized.

Extensions to `math` are likely, such as recognizing `_` and `^` for subscripts and superscripts.

11.7 Links

```
(secref tag
  [#:doc module-path
   #:underline? underline?]) → element?
tag : string?
module-path : (or/c module-path? false/c) = #f
underline? : any/c = #t
```

Inserts the hyperlinked title of the section tagged *tag*, but "aux-element" items in the title content are omitted in the hyperlink label.

If *module-path* is provided, the *tag* refers to a tag with a prefix determined by *module-path*. When `setup-plt` renders documentation, it automatically adds a tag prefix to the document based on the source module. Thus, for example, to refer to a section of the PLT Scheme reference, *module-path* would be `'(lib "scribblings/reference/reference.scrbl")`.

If *underline?* is `#f`, then the hyperlink is rendered in HTML without an underline.

```
(seclink tag
  [#:doc module-path
   #:underline? underline?]
  pre-content ...) → element?
tag : string?
module-path : (or/c module-path? false/c) = #f
underline? : any/c = #t
pre-content : any/c
```

Like `secref`, but the link label is the decoded `pre-content` instead of the target section's name.

```
(other-manual module-path
  [#:underline? underline?]) → element?
  module-path : module-path?
  underline? : any/c = #t
```

Like `secref` for the document's implicit "top" tag. Use this function to refer to a whole manual instead of `secref`, in case a special style in the future is used for manual titles.

```
(schemelink id pre-content ...) → element?
  id : symbol?
  pre-content : any/c
```

The decoded `pre-content` is hyperlinked to the definition of `id`.

```
(link url
  pre-content ...
  [#:underline? underline?
   #:style style]) → element?
  url : string?
  pre-content : any/c
  underline? : any/c = #t
  style : any/c = (if underline? #f "plainlink")
```

The decoded `pre-content` is hyperlinked to `url`. If `style` is not supplied, then `underline?` determines how the link is rendered.

```
(elemtag t pre-content ...) → element?
  t : tag?
  pre-content : any/c
```

The tag `t` refers to the content form of `pre-content`.

```
(elemref t pre-content ...) → element?
  t : tag?
  pre-content : any/c
```

The decoded `pre-content` is hyperlinked to `t`, which is normally defined using `elemtag`.

```
(deftech pre-content ...) → element?  
pre-content : any/c
```

Produces an element for the decoded *pre-content*, and also defines a term that can be referenced elsewhere using *tech*.

The *content*->*string* result of the decoded *pre-content* is used as a key for references, but normalized as follows:

- A trailing “ies” is replaced by “y”.
- A trailing “s” is removed.
- Consecutive hyphens and whitespaces are all replaced by a single space.

These normalization steps help support natural-language references that differ slightly from a defined form. For example, a definition of “bananas” can be referenced with a use of “banana”.

```
(tech pre-content ... [#:doc module-path]) → element?  
pre-content : any/c  
module-path : (or/c module-path? false/c) = #f
```

Produces an element for the decoded *pre-content*, and hyperlinks it to the definition of the content as established by *deftech*. The content’s string form is normalized in the same way as for *deftech*. The *#:doc* argument supports cross-document references, like in *secref*.

The hyperlink is relatively quiet, in that underlining in HTML output appears only when the mouse is moved over the term.

In some cases, combining both natural-language uses of a term and proper linking can require some creativity, even with the normalization performed on the term. For example, if “bind” is defined, but a sentence uses the term “binding,” the latter can be linked to the former using `@tech{bind}ing`.

```
(techlink pre-content ... [#:doc module-path]) → element?  
pre-content : any/c  
module-path : (or/c module-path? false/c) = #f
```

Like *tech*, but the link is not a quiet. For example, in HTML output, a hyperlink underline appears even when the mouse is not over the link.

11.8 Indexing

```
(indexed-scheme datum ...)
```

A combination of `scheme` and `as-index`, with the following special cases when a single `datum` is provided:

- If `datum` is a quote form, then the quote is removed from the key (so that it's sorted using its unquoted form).
- If `datum` is a string, then quotes are removed from the key (so that it's sorted using the string content).

```
(idefterm pre-content ...) → element?  
pre-content : any/c
```

Combines `as-index` and `defterm`. The content normally should be plural, rather than singular. Consider using `deftech`, instead, which always indexes.

```
(pidefterm pre-content ...) → element?  
pre-content : any/c
```

Like `idefterm`, but plural: adds an “s” on the end of the content for the index entry. Consider using `deftech`, instead.

```
(indexed-file pre-content ...) → element?  
pre-content : any/c
```

A combination of `file` and `as-index`, but where the sort key for the index item does not include quotes.

```
(indexed-envvar pre-content ...) → element?  
pre-content : any/c
```

A combination of `envvar` and `as-index`.

11.9 Images

```
(image filename-relative-to-source
  pre-element ...) → flow-element?
  filename-relative-to-source : string?
  pre-element : any/c
```

Creates a centered image from the given relative source path. The decoded `pre-content` serves as the alternate text for contexts where the image cannot be displayed.

The path is relative to the current directory, which is set by `setup-plt` and `scribble` to the directory of the main document file.

```
(image/plain filename-relative-to-source
  pre-element ...) → element?
  filename-relative-to-source : string?
  pre-element : any/c
```

Like `image`, but the result is an element to appear inline in a paragraph.

11.10 Bibliography

```
(cite key) → element?
  key : string?
```

Links to a bibliography entry, using `key` both to indicate the bibliography entry and, in square brackets, as the link text.

```
(bibliography #:tag string? entry ...) → part?
  string? : "doc-bibliography"
  entry : bib-entry?
```

Creates a bibliography part containing the given entries, each of which is created with `bib-entry`. The entries are typeset in order as given

```
(bib-entry #:key key
  #:title title
  [#:is-book? is-book?]
  #:author author
  #:location location
  #:date date
  [#:url url]) → bib-entry?
  key : string?
```

```
title : any/c
is-book? : any/c = #f
author : any/c
location : any/c
date : any/c
url : any/c = #f
```

Creates a bibliography entry. The *key* is used to refer to the entry via `cite`. The other arguments are used as elements in the entry:

- *title* is the title of the cited work. It will be surrounded by quotes in typeset form if *is-book?* is `#f`, otherwise it is typeset via *italic*.
- *author* lists the authors. Use names in their usual order (as opposed to “last, first”), and separate multiple names with commas using “and” before the last name (where there are multiple names). The *author* is typeset in the bibliography as given.
- *location* names the publication venue, such as a conference name or a journal with volume, number, and pages. The *location* is typeset in the bibliography as given.
- *date* is a date, usually just a year (as a string). It is typeset in the bibliography as given.
- *url* is an optional URL. It is typeset in the bibliography using `tt` and hyperlinked.

```
(bib-entry? v) → boolean?
v : any/c
```

Returns `#t` if *v* is a bibliography entry created by `bib-entry`, `#f` otherwise.

11.11 Miscellaneous

```
(t pre-content ...) → paragraph?
pre-content : any/c
```

Wraps the decoded *pre-content* as a paragraph.

```
PLaneT : string?
```

"PLaneT" (to help make sure you get the letters in the right case).

`(hash-lang)` → `element?`

Returns an element for `#lang` that is hyperlinked to an explanation.

`void-const` : `element?`

Returns an element for `#<void>`.

`undefined-const` : `element?`

Returns an element for `#<undefined>`.

`(centerline pre-flow ...)` → `table?`
`pre-flow` : `any/c`

Produces a centered table with the `pre-flow` parsed by `decode-flow`.

`(commandline pre-content ...)` → `paragraph?`
`pre-content` : `any/c`

Produces an inset command-line example (e.g., in typewriter font).

`(margin-note pre-content ...)` → `paragraph?`
`pre-content` : `any/c`

Produces a paragraph to be typeset in the margin instead of inlined.

11.12 Index-Entry Descriptions

`(require scribble/manual-struct)`

The `scribble/manual-struct` library provides types used to describe index entries created by `scribble/manual` functions. These structure types are provided separate from `scribble/manual` so that `scribble/manual` need not be loaded when deserializing cross-reference information that was generated by a previously rendered document.

`(struct module-path-index-desc ())`

Indicates that the index entry corresponds to a module definition via `defmodule` and `company`.

```
(struct exported-index-desc (name from-libs))
  name : symbol?
  from-libs : (listof module-path?)
```

Indicates that the index entry corresponds to the definition of an exported binding. The `name` field and `from-libs` list correspond to the documented name of the binding and the primary modules that export the documented name (but this list is not exhaustive, because new modules can re-export the binding).

```
(struct (form-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of a syntactic form via `defform` and `company`.

```
(struct (procedure-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of a procedure binding via `defproc` and `company`.

```
(struct (thing-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of a binding via `defthing` and `company`.

```
(struct (struct-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of a structure type via `defstruct` and `company`.

```
(struct (class-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of a class via `defclass` and `company`.

```
(struct (interface-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of an interface via `definterface` and `company`.

```
(struct (mixin-index-desc exported-index-desc) ())
```

Indicates that the index entry corresponds to the definition of a mixin via `defmixin` and `company`.

```
(struct (method-index-desc exported-index-desc) (method-name
                                                class-tag))

method-name : symbol?
class-tag   : tag?
```

Indicates that the index entry corresponds to the definition of an method via `defmethod` and `company`. The *name* field from `exported-index-desc` names the class or interface that contains the method. The `method-name` field names the method. The `class-tag` field provides a pointer to the start of the documentation for the method's class or interface.

12 Evaluation and Examples

```
(require scribble/eval)
```

The `scribble/eval` library provides utilities for evaluating code at document-build time and incorporating the results in the document, especially to show example uses of defined procedures and syntax.

```
(interaction datum ...)  
(interaction #:eval eval-expr datum ...)
```

Like `schemeinput`, except that the result for each input `datum` is shown on the next line. The result is determined by evaluating the quoted form of the `datum` using the evaluator produced by `eval-expr`, if provided.

The `eval-expr` must produce a sandbox evaluator via `make-evaluator` or `make-module-evaluator` with the `sandbox-output` and `sandbox-error-output` parameters set to `'string`. If `eval` is not provided, an evaluator is created using `make-base-eval`.

Uses of `code:comment` and `code:blank` are stripped from each `datum` before evaluation.

If a `datum` has the form `(eval:alts show-datum eval-datum)`, then `show-datum` is typeset, while `eval-datum` is evaluated.

```
(interaction-eval datum)  
(interaction-eval #:eval eval-expr datum)
```

Like `interaction`, evaluates the quoted form of `datum`, but returns the empty string.

```
(interaction-eval-show datum)  
(interaction-eval-show #:eval eval-expr datum)
```

Like `interaction-eval`, but produces an element representing the printed form of the evaluation result.

```
(schemeblock+eval datum ...)  
(schemeblock+eval #:eval eval-expr datum ...)
```

Combines `schemeblock` and `interaction-eval`.

```
(schememod+eval name datum ...)  
(schememod+eval #:eval eval-expr name datum ...)
```

Combines `schememod` and `interaction-eval`.

```
(def+int defn-datum expr-datum ...)  
(def+int #:eval eval-expr defn-datum expr-datum ...)
```

Like `interaction`, except the the *defn-datum* is typeset as for `schemeblock` (i.e., no prompt) and a line of space is inserted before the *expr-datums*.

```
(defs+int (defn-datum ...) expr-datum ...)  
(defs+int #:eval eval-expr (defn-datum ...) expr-datum ...)
```

Like `def+int`, but for multiple leading definitions.

```
(examples datum ...)  
(examples #:eval eval-expr datum ...)
```

Like `interaction`, but with an “Examples:” label prefixed.

```
(defexamples datum ...)  
(defexamples #:eval eval-expr datum ...)
```

Like `examples`, but each definition using `define` or `define-struct` among the *datums* is typeset without a prompt, and with line of space after it.

```
(make-base-eval) → (any/c . -> . any)
```

Creates an evaluator using `(make-evaluator 'scheme/base)`, setting sandbox parameters to disable limits, set the outputs to `'string`, and not add extra security guards.

```
(close-eval eval) → (one-of/c "")  
  eval : (any/c . -> . any)
```

Shuts down an evaluator produced by `make-base-eval`. Use `close-eval` when garbage collection cannot otherwise reclaim an evaluator (e.g., because it is defined in a module body).

```
(scribble-eval-handler)  
  → ((any/c . -> . any) any/c boolean? . -> . any)  
(scribble-eval-handler handler) → void?  
  handler : ((any/c . -> . any) any/c boolean? . -> . any)
```

A parameter that serves as a hook for evaluation. The evaluator to use is supplied as the first argument to the parameter's value, and the second argument is the form to evaluate. The last argument is `#t` if exceptions are being captured (to display exception results), `#f` otherwise.

13 BNF Grammars

```
(require scribble/bnf)
```

The `scribble/bnf` library provides utilities for typesetting grammars.

See also `schemegrammar`.

```
(BNF prod ...) → table?  
  prod : (cons element? (listof element?))
```

Typesets a grammar table. Each production starts with an element (typically constructed with `nonterm`) for the non-terminal being defined, and then a list of possibilities (typically constructed with `BNF-seq`, etc.) to show on separate lines.

```
(nonterm pre-content ...) → element?  
  pre-content : any/c
```

Typesets a non-terminal: italic in angle brackets.

```
(BNF-seq elem ...) → element?  
  elem : element?
```

Typesets a sequence.

```
(BNF-group pre-content ...) → element?  
  pre-content : any/c
```

Typesets a group surrounded by curly braces (so the entire group can be repeated, for example).

```
(optional pre-content ...) → element?  
  pre-content : any/c
```

Typesets an optional element: in square brackets.

```
(kleenestar pre-content ...) → element?  
  pre-content : any/c
```

Typesets a 0-or-more repetition.

```
(kleeneplus pre-content ...) → element?  
pre-content : any/c
```

Typesets a 1-or-more repetition.

```
(kleenerange n m pre-content ...) → element?  
n : any/c  
m : any/c  
pre-content : any/c
```

Typesets a *n*-to-*m* repetition. The *n* and *m* arguments are converted to a string using `(format "~a" n)` and `(format "~a" m)`.

```
(BNF-alt elem ...) → element?  
elem : element?
```

Typesets alternatives for a production's right-hand side to appear on a single line. The result is normally used as a single possibility in a production list for [BNF](#).

```
BNF-etc : string?
```

A string to use for omitted productions or content.

14 Cross-Reference Utilities

```
(require scribble/xref)
```

The `scribble/xref` library provides utilities for querying cross-reference information that was collected from a document build.

```
(xref? v) → boolean?  
v : any/c
```

Returns `#t` if `v` is a cross-reference record created by `load-xref`, `#f` otherwise.

```
(load-xref sources  
  [#:render% using-render%  
   #:root root-path]) → xref?  
sources : (listof (-> any/c))  
using-render% : (subclass?/c render%) = (render-mixin render%)  
root-path : (or/c path-string? false/c) = #f
```

Creates a cross-reference record given a list of functions that each produce a serialized information obtained from `serialize-info` in `render%`. If a `sources` element produces `#f`, its result is ignored.

Since the format of serialized information is specific to a rendering class, the optional `using-render%` argument accepts the relevant class. It default to HTML rendering.

If `root-path` is not `#f`, then file paths that are serialized as relative to an instantiation-supplied `root-path` are deserialized as relative instead to the given `root-path`.

Use `load-collections-xref` from `setup/xref` to get all cross-reference information for installed documentation.

```
(xref-binding->definition-tag xref  
  binding  
  mode) → (or/c tag? false/c)  
xref : xref?
```

```

binding : (or/c identifier?
           (list/c (or/c module-path?
                    module-path-index?)
                   symbol?))
           (listof module-path-index?
                   symbol?
                   module-path-index?
                   symbol?
                   (one-of/c 0 1)
                   (or/c exact-integer? false/c)
                   (or/c exact-integer? false/c)))
mode : (or/c exact-integer? false/c)

```

Locates a tag in *xref* that documents a module export. The binding is specified in one of several ways, as described below; all possibilities encode an exporting module and a symbolic name. The name must be exported from the specified module. Documentation is found either for the specified module or, if the exported name is re-exported from other other module, for the other module (transitively).

The *mode* argument specifies the relevant phase level for the binding. The *binding* is specified in one of four ways:

- If *binding* is an identifier, then *identifier-binding* is used with *mode* to determine the binding.
- If *binding* is a two-element list, then the first element provides the exporting module and the second the exported name. The *mode* argument is effectively ignored.
- If *binding* is a seven-element list, then it corresponds to a result from *identifier-binding* using *mode*.
- If *binding* is a five-element list, then the first element is as for the two-element-list case, and the remain elements are as in the last four elements of the seven-element case.

If a documentation point exists in *xref*, a tag is returned, which might be used with *xref-tag->path+anchor* or embedded in a document rendered via *xref-render*. If no definition point is found in *xref*, the result is *#f*.

```

(xref-tag->path+anchor xref
                      tag
                      [#:render% using-render%])
→ (or/c false/c path?)
   (or/c false/c string?)
xref : xref?
tag : tag?
using-render% : (subclass?/c render%) = (render-mixin render%)

```

Returns a path and anchor string designated by the key *tag* according the cross-reference *xref*. The first result is *#f* if no mapping is found for the given tag. The second result is *#f* if the first result is *#f*, and it can also be *#f* if the tag refers to a page rather than a specific point in a page.

The optional *using-render%* argument is as for *load-xref*.

```
(xref-tag->index-entry xref tag) → (or/c false/c entry?)
  xref : xref?
  tag  : tag?
```

Extract an *entry* structure that provides addition information about the definition (of any) referenced by *tag*. This function can be composed with *xref-binding->definition-tag* to obtain information about a binding, such as the library that exports the binding and its original name.

```
(xref-render xref
             doc
             dest
             [#:render% using-render%
              #:refer-to-existing-files? use-existing?])
→ (or/c void? any/c)
  xref : xref?
  doc  : part?
  dest : (or/c path-string? false/c)
  using-render% : (subclass?/c render%) = (render-mixin render%)
  use-existing? : any/c = (not dest)
```

Renders *doc* using the cross-reference info in *xref* to the destination *dest*. For example, *doc* might be a generated document of search results using link tags described in *xref*.

If *dest* is *#f*, no file is written, and the result is an X-expression for the rendered page. Otherwise, the file *dest* is written and the result is *#<void>*.

The optional *using-render%* argument is as for *load-xref*. It determines the kind of output that is generated.

If *use-existing?* is true, then files referenced during rendering (such as image files) are referenced from their existing locations, instead of copying to the directory of *dest*.

```
(xref-transfer-info renderer ci xref) → void?
  renderer : (is-a?/c render%)
  ci       : collect-info?
  xref     : xref?
```

Transfers cross-reference information to *ci*, which is the initially collected information from *renderer*.

```
(xref-index xref) → (listof entry?)  
  xref : xref?
```

Converts indexing information *xref* into a list of *entry* structures.

```
(struct entry (words content tag desc))  
  words : (and/c (listof string?) cons?)  
  content : list?  
  tag : tag?  
  desc : any/c
```

Represents a single entry in a Scribble document index.

The *words* list corresponds to [index-element-plain-seq](#). The *content* list corresponds to [index-element-entry-seq](#). The *desc* value corresponds to [index-element-desc](#). The *tag* is the destination for the index link into the main document.

15 Text Preprocessor

```
#lang scribble/text
```

The `scribble/text` language provides everything from `scheme/base` with a few changes that make it suitable as a preprocessor language:

- It uses `read-syntax-inside` to read the body of the module, similar to §8 “Document Reader”.
- It has a custom printer (`current-print`) that displays all values. The printer is also installed as the `port-display-handler` so it can be used through `display` as well as `~a` in format strings. The printer displays most values (as is usual for `display`), except for
 - `void` and `#f` are not displayed,
 - pairs are displayed recursively (just their contents, no parentheses),
 - promises are forced, thunks are invoked.

This means that to write a text file that has scheme code, you simply write it as a module in the `scribble/text` language, and run it through `mzscheme`. Here is a sample file:

```
#lang scribble/text
@(define (angled . body) (list "<" body ">"))@;
@(define (shout . body) @angled[(map string-upcase body)])@;
blah @angled{blah @shout{blah} blah} blah
```

(Note how `@;` is used to avoid empty lines in the output.)

15.1 Using External Files

Using additional files that contain code for your preprocessing is trivial: the preprocessor source is a plain Scheme file, so you can `require` additional files as usual.

However, things can become tricky if you want to include an external file that should also be preprocessed. Using `require` with a text file (that uses the `scribble/text` language) almost works, but when a module is required, it is invoked before the current module, which means that the required file will be preprocessed before the current file regardless of where the `require` expression happens to be. Alternatively, you can use `dynamic-require` with `#f` for the last argument (which makes it similar to a plain `load`)—but remember that the path will be relative to the current directory, not to the source file.

Finally, there is a convenient syntax for including text files to be processed:

```
(include filename)
```

Preprocess the *filename* using the same syntax as `scribble/text`. This is similar to using `load` in a namespace that can access names bound in the current file so included code can refer to bindings from the including module. Note, however, that the including module cannot refer to names that are bound the included file because it is still a plain scheme module—for such uses you should still use `require` as usual.

Index

@-Reader, 20
as-index
aux-elem, 60
aux-element, 41
aux-element?, 41
Base Renderer
Basic Document Forms, 57
bib-entry, 84
bib-entry?, 85
Bibliography, 84
bibliography, 84
block, 34
block-width, 44
block?, 43
blockquote, 39
blockquote, 34
blockquote-paragraphs, 39
blockquote-style, 39
blockquote?, 39
BNF, 92
BNF Grammars, 92
BNF-alt, 93
BNF-etc, 93
BNF-group, 92
BNF-seq, 92
bold, 60
centerline
cite, 84
class-index-desc, 87
class-index-desc?, 87
clean-up-index-string, 53
close-eval, 90
collect, 48
collect pass, 34
collect-element, 42
collect-element-collect, 42
collect-element?, 42
collect-info, 44
collect-info-ext-ht, 44
collect-info-gen-prefix, 44
collect-info-ht, 44
collect-info-parents, 44
collect-info-parts, 44
collect-info-relatives, 44
collect-info-tags, 44
collect-info?, 44
collect-put!, 45
Collected and Resolved Information, 36
collected-info, 42
collected-info-info, 42
collected-info-number, 42
collected-info-parent, 42
collected-info?, 42
commandline, 86
Concrete Syntax, 20
content, 34
content->string, 43
Cross-Reference Utilities, 94
declare-exporting
decode, 51
decode, 51
decode-content, 52
decode-elements, 52
decode-flow, 52
decode-paragraph, 52
decode-part, 51
decode-string, 52
Decoding Text, 51
def+int, 90
defboolparam, 74
defclass, 75
defclass/title, 75
defconstructor, 76
defconstructor*/make, 76
defconstructor/auto-super, 76
defconstructor/make, 76
defexamples, 90
deform, 72
deform*, 72
deform*/subs, 73
deform/none, 73
deform/subs, 72

- defidform, 73
- Defining Scheme Bindings, 9
- definterface, 76
- definterface/title, 76
- defmethod, 77
- defmethod*, 77
- defmixin, 76
- defmixin/title, 76
- defmodule, 68
- defmodule*, 69
- defmodule*/no-declare, 69
- defmodulelang, 68
- defmodulelang*, 69
- defmodulelang*/no-declare, 69
- defparam, 74
- defproc, 70
- defproc*, 72
- defs+int, 90
- defsignature, 77
- defsignature/splice, 77
- defstruct, 74
- deftech, 82
- defterm, 78
- defthing, 74
- deftogether, 74
- delayed block*, 35
- delayed element*, 35
- delayed-block, 39
- delayed-block-resolve, 39
- delayed-block?, 39
- delayed-element, 41
- delayed-element-plain, 41
- delayed-element-resolve, 41
- delayed-element-sizer, 41
- delayed-element?, 41
- deserialize-info, 48
- DFlag, 79
- Document Language, 55
- Document Reader, 56
- Document Structure, 57
- Document Syntax, 6
- Documenting Classes and Interfaces, 75

- Documenting Forms, Functions, Structure
Types, and Values, 70
- Documenting Modules, 68
- Documenting Signatures, 77
- DFlag, 79
- elem
- element, 39
- element*, 34
- element->string, 43
- element-content, 39
- element-style, 39
- element-width, 44
- element?, 39
- elemref, 81
- elemtag, 81
- emph, 78
- entry, 97
- entry-content, 97
- entry-desc, 97
- entry-tag, 97
- entry-words, 97
- entry?, 97
- envvar, 79
- Evaluation and Examples, 89
- examples, 90
- exec, 79
- exported-index-desc, 87
- exported-index-desc-from-libs, 87
- exported-index-desc-name, 87
- exported-index-desc?, 87
- filepath
- Flag, 79
- flow, 38
- flow*, 34
- flow-paragraphs, 38
- flow?, 38
- form-index-desc, 87
- form-index-desc?, 87
- generated-tag
- generated-tag?, 43
- Getting Started, 5
- hash-lang

[hover-element](#), 41
[hover-element-text](#), 41
[hover-element?](#), 41
 How to Scribble Documentation, 5
[hspace](#), 60
 HTML Renderer, 49
[idefterm](#)
[image](#), 84
[image-file](#), 42
[image-file-path](#), 42
[image-file-scale](#), 42
[image-file?](#), 42
[image/plain](#), 84
 Images, 83
[include](#), 99
[include-section](#), 59
[index](#), 61
[index*](#), 61
[index-element](#), 40
[index-element-desc](#), 40
[index-element-entry-seq](#), 40
[index-element-plain-seq](#), 40
[index-element-tag](#), 40
[index-element?](#), 40
 Index-Entry Descriptions, 86
[index-section](#), 61
[indexed-envvar](#), 83
[indexed-file](#), 83
[indexed-scheme](#), 83
 Indexing, 61
 Indexing, 83
info key, 44
[info-key?](#), 44
 interaction, 89
[interaction-eval](#), 89
[interaction-eval-show](#), 89
 Interface, 31
[interface-index-desc](#), 87
[interface-index-desc?](#), 87
italic, 60
[item](#), 59
[item?](#), 59
[itemization](#), 38
itemization, 34
[itemization-flows](#), 38
[itemization?](#), 38
[itemize](#), 59
 Javascript
[kleeneplus](#)
[kleenerange](#), 93
[kleenestar](#), 92
 Latex Renderer
 Layer Roadmap, 18
[link](#), 81
[link-element](#), 40
[link-element-tag](#), 40
[link-element?](#), 40
 Links, 80
[litchar](#), 66
[load-xref](#), 94
[local-table-of-contents](#), 62
[make-at-readtable](#)
[make-aux-element](#), 41
[make-base-eval](#), 90
[make-blockquote](#), 39
[make-class-index-desc](#), 87
[make-collect-element](#), 42
[make-collect-info](#), 44
[make-collected-info](#), 42
[make-delayed-block](#), 39
[make-delayed-element](#), 41
[make-element](#), 39
[make-entry](#), 97
[make-exported-index-desc](#), 87
[make-flow](#), 38
[make-form-index-desc](#), 87
[make-generated-tag](#), 43
[make-hover-element](#), 41
[make-image-file](#), 42
[make-index-element](#), 40
[make-interface-index-desc](#), 87
[make-itemization](#), 38
[make-link-element](#), 40
[make-method-index-desc](#), 88

- [make-mixin-index-desc](#), 87
- [make-module-path-index-desc](#), 86
- [make-paragraph](#), 38
- [make-part](#), 37
- [make-part-collect-decl](#), 53
- [make-part-index-decl](#), 53
- [make-part-relative-element](#), 41
- [make-part-start](#), 53
- [make-part-tag-decl](#), 53
- [make-procedure-index-desc](#), 87
- [make-resolve-info](#), 44
- [make-script-element](#), 41
- [make-splice](#), 53
- [make-struct-index-desc](#), 87
- [make-styled-paragraph](#), 38
- [make-table](#), 38
- [make-target-element](#), 40
- [make-target-url](#), 42
- [make-thing-index-desc](#), 87
- [make-title-decl](#), 52
- [make-toc-element](#), 40
- [make-toc-target-element](#), 40
- [make-unnumbered-part](#), 38
- [make-versioned-part](#), 38
- [make-with-attributes](#), 42
- Manual Forms, 64
- [margin-note](#), 86
- [math](#), 80
- [menutitem](#), 78
- [method](#), 77
- [method-index-desc](#), 88
- [method-index-desc-class-tag](#), 88
- [method-index-desc-method-name](#), 88
- [method-index-desc?](#), 88
- Miscellaneous, 85
- [mixin-index-desc](#), 87
- [mixin-index-desc?](#), 87
- [module-path-index-desc](#), 86
- [module-path-index-desc?](#), 86
- [module-path-prefix->string](#), 59
- Multi-Page Sections, 12
- [nonterm](#)
- [onscreen](#)
- [optional](#), 92
- [other-manual](#), 81
- [paragraph](#)
- [paragraph](#), 34
- [paragraph-content](#), 38
- [paragraph?](#), 38
- [part](#), 37
- [part](#), 34
- [part-collect-decl](#), 53
- [part-collect-decl-element](#), 53
- [part-collect-decl?](#), 53
- [part-collected-info](#), 46
- [part-flow](#), 37
- [part-index-decl](#), 53
- [part-index-decl-entry-seq](#), 53
- [part-index-decl-plain-seq](#), 53
- [part-index-decl?](#), 53
- [part-parts](#), 37
- [part-relative element](#), 35
- [part-relative-element](#), 41
- [part-relative-element-plain](#), 41
- [part-relative-element-resolve](#), 41
- [part-relative-element-sizer](#), 41
- [part-relative-element?](#), 41
- [part-start](#), 53
- [part-start-depth](#), 53
- [part-start-style](#), 53
- [part-start-tag-prefix](#), 53
- [part-start-tags](#), 53
- [part-start-title](#), 53
- [part-start?](#), 53
- [part-style](#), 37
- [part-tag-decl](#), 53
- [part-tag-decl-tag](#), 53
- [part-tag-decl?](#), 53
- [part-tag-prefix](#), 37
- [part-tags](#), 37
- [part-title-content](#), 37
- [part-to-collect](#), 37
- [part?](#), 37
- Parts, 34

[PFlag](#), 79
[pidefterm](#), 83
[PLaneT](#), 85
[procedure](#), 68
[procedure-index-desc](#), 87
[procedure-index-desc?](#), 87
[Prose and Terminology](#), 13
[read](#)
[read-inside](#), 32
[read-syntax](#), 32
[read-syntax-inside](#), 32
[render](#), 48
render pass, 34
[render%](#), 47
[render-mixin](#), 49
[render-mixin](#), 49
[render-mixin](#), 49
[render-multi-mixin](#), 49
[Renderer](#), 47
[resolve](#), 48
resolve pass, 34
[resolve-get](#), 45
[resolve-get-keys](#), 46
[resolve-get/ext?](#), 45
[resolve-get/tentative](#), 46
[resolve-info](#), 44
[resolve-info-ci](#), 44
[resolve-info-delays](#), 44
[resolve-info-undef](#), 44
[resolve-info?](#), 44
[resolve-search](#), 45
[Scheme](#)
[SCHEME](#), 66
[scheme](#), 66
[Scheme Typesetting and Hyperlinks](#), 7
[schemeblock](#), 64
[SCHEMEBLOCK](#), 65
[schemeblock+eval](#), 89
[SCHEMEBLOCK0](#), 65
[schemeblock0](#), 65
[schemeerror](#), 67
[schemefont](#), 66
[schemegrammar](#), 75
[schemegrammar*](#), 75
[schemeid](#), 66
[schemeidfont](#), 67
[schemeinput](#), 65
[schemekeywordfont](#), 67
[schemelink](#), 81
[schememetafont](#), 67
[schememod](#), 65
[schememod+eval](#), 89
[schememodname](#), 66
[schemeparenfont](#), 67
[schemeresult](#), 66
[schemeresultfont](#), 67
[schemevalfont](#), 67
[schemevarfont](#), 67
[Scribble Layers](#), 16
[scribble-eval-handler](#), 90
[scribble/base-render](#), 47
[scribble/basic](#), 57
[scribble/bnf](#), 92
[scribble/decode](#), 51
[scribble/doc](#), 56
[scribble/doclang](#), 55
[scribble/eval](#), 89
[scribble/html-render](#), 49
[scribble/latex-render](#), 49
[scribble/manual](#), 64
[scribble/manual-struct](#), 86
[scribble/reader](#), 31
[scribble/scheme](#), 63
[scribble/struct](#), 34
[scribble/text](#), 98
[scribble/text-render](#), 49
[scribble/xref](#), 94
Scribble: PLT Documentation Tool, 1
[script-element](#), 41
[script-element-script](#), 41
[script-element-type](#), 41
[script-element?](#), 41
[seclink](#), 80
[secref](#), 80

[section](#), 58
[Section Hyperlinks](#), 8
[Section Titles](#), 15
[section-index](#), 61
[serialize-info](#), 48
[set-external-tag-path](#), 49
[Showing Scheme Examples](#), 11
[sigelem](#), 78
[signature-desc](#), 78
[span-class](#), 60
[specform](#), 73
[specspectsubform](#), 73
[specspectsubform/subs](#), 74
[specsform](#), 73
[specsform/subs](#), 73
[splice](#), 53
[splice-run](#), 53
[splice?](#), 53
[Splitting the Document Source](#), 11
[struct-index-desc](#), 87
[struct-index-desc?](#), 87
[struct:aux-element](#), 41
[struct:blockquote](#), 39
[struct:class-index-desc](#), 87
[struct:collect-element](#), 42
[struct:collect-info](#), 44
[struct:collected-info](#), 42
[struct:delayed-block](#), 39
[struct:delayed-element](#), 41
[struct:element](#), 39
[struct:entry](#), 97
[struct:exported-index-desc](#), 87
[struct:flow](#), 38
[struct:form-index-desc](#), 87
[struct:generated-tag](#), 43
[struct:hover-element](#), 41
[struct:image-file](#), 42
[struct:index-element](#), 40
[struct:interface-index-desc](#), 87
[struct:itemization](#), 38
[struct:link-element](#), 40
[struct:method-index-desc](#), 88
[struct:mixin-index-desc](#), 87
[struct:module-path-index-desc](#), 86
[struct:paragraph](#), 38
[struct:part](#), 37
[struct:part-collect-decl](#), 53
[struct:part-index-decl](#), 53
[struct:part-relative-element](#), 41
[struct:part-start](#), 53
[struct:part-tag-decl](#), 53
[struct:procedure-index-desc](#), 87
[struct:resolve-info](#), 44
[struct:script-element](#), 41
[struct:splice](#), 53
[struct:struct-index-desc](#), 87
[struct:styled-paragraph](#), 38
[struct:table](#), 38
[struct:target-element](#), 40
[struct:target-url](#), 42
[struct:thing-index-desc](#), 87
[struct:title-decl](#), 52
[struct:toc-element](#), 40
[struct:toc-target-element](#), 40
[struct:unnumbered-part](#), 38
[struct:versioned-part](#), 38
[struct:with-attributes](#), 42
[Structure Reference](#), 37
[Structures And Processing](#), 34
[Style Guide](#), 13
[styled-paragraph](#), 38
[styled-paragraph-style](#), 38
[styled-paragraph?](#), 38
[subscript](#), 60
[subsection](#), 58
[subsubsection](#), 58
[subsubsub*section](#), 58
[superscript](#), 60
[svar](#), 68
[Syntax Properties](#), 30
[t](#)
[table](#), 38
[table](#), 34
[table-flowss](#), 38

[table-of-contents](#), 62
[table-style](#), 38
[table?](#), 38
 Tables of Contents, 62
tag, 35
tag prefix, 36
[tag-key](#), 46
[tag?](#), 43
 Tags, 35
[target-element](#), 40
[target-element-tag](#), 40
[target-element?](#), 40
[target-url](#), 42
[target-url-addr](#), 42
[target-url-style](#), 42
[target-url?](#), 42
tech, 82
[techlink](#), 82
 Text Preprocessor, 98
 Text Renderer, 49
 Text Styles, 59
 The Body Part, 24
 The Command Part, 22
 The Datum Part, 23
[thing-index-desc](#), 87
[thing-index-desc?](#), 87
title, 57
[title-decl](#), 52
[title-decl-content](#), 52
[title-decl-style](#), 52
[title-decl-tag-prefix](#), 52
[title-decl-tags](#), 52
[title-decl-version](#), 52
[title-decl?](#), 52
[toc-element](#), 40
[toc-element-toc-content](#), 40
[toc-element?](#), 40
[toc-target-element](#), 40
[toc-target-element?](#), 40
tt, 60
 Typesetting Code, 14
 Typesetting Code, 64
 Typesetting Prose, 15
 Typical Composition, 16
[undefined-const](#)
[unnumbered-part](#), 38
[unnumbered-part?](#), 38
[use-at-readtable](#), 33
 Using External Files, 98
var
 Various String Forms, 78
verbatim, 66
[versioned-part](#), 38
[versioned-part-version](#), 38
[versioned-part?](#), 38
[void-const](#), 86
[whitespace?](#)
[with-attributes](#), 42
[with-attributes-assoc](#), 42
[with-attributes-style](#), 42
[with-attributes?](#), 42
xmethod
[xref-binding->definition-tag](#), 94
[xref-index](#), 97
[xref-render](#), 96
[xref-tag->index-entry](#), 96
[xref-tag->path+anchor](#), 95
[xref-transfer-info](#), 96
[xref?](#), 94