

Scribble: The Racket Documentation Tool

Version 6.1.1

Matthew Flatt
and Eli Barzilay

November 4, 2014

Scribble is a collection of tools for creating prose documents—papers, books, library documentation, etc.—in HTML or PDF (via Latex) form. More generally, Scribble helps you write programs that are rich in textual content, whether the content is prose to be typeset or any other form of text to be generated programmatically.

This document is itself written using Scribble. You can see its source at <http://git.racket-lang.org/plt/tree/HEAD:/pkgs/scribble-pkgs/scribble-doc/scriblings/scribble>, starting with the "scribble.scrbl" file.

Contents

1 Getting Started	6
1.1 A First Example	6
1.2 Multiple Sections	7
1.3 Splitting the Document Source	7
1.4 Document Styles	8
1.5 More Functions	9
1.5.1 Centering	10
1.5.2 Margin Notes	10
1.5.3 Itemizations	11
1.5.4 Tables	11
1.6 Text Mode vs. Racket Mode for Arguments	11
1.7 @ Syntax Basics	13
1.8 Decoding Sequences	15
1.9 Pictures	16
1.10 Next Steps	16
2 @ Syntax	17
2.1 The Scribble Syntax at a Glance	17
2.2 The Command Part	21
2.3 The Datum Part	22
2.4 The Body Part	23
2.4.1 Alternative Body Syntax	24
2.4.2 Racket Expression Escapes	24
2.4.3 Comments	25
2.4.4 Spaces, Newlines, and Indentation	26
3 High-Level Scribble API	30
3.1 Base Document Format	30
3.1.1 Document Structure	30
3.1.2 Blocks	33
3.1.3 Text Styles and Content	37
3.1.4 Spacing	39
3.1.5 Links	40
3.1.6 Indexing	43
3.1.7 Tables of Contents	44
3.1.8 Tags	44
3.2 Racket Manual Format	44
3.3 Book Format	45
3.4 Report Format	45
3.5 SIGPLAN Paper Format	45
3.6 JFP Paper Format	48
3.7 LNCS Paper Format	49
3.7.1 Example	50

4 Scribbling Documentation	51
4.1 Getting Started with Documentation	51
4.1.1 Setting Up Library Documentation	51
4.1.2 Racket Typesetting and Hyperlinks	52
4.1.3 Section Hyperlinks	53
4.1.4 Defining Racket Bindings	54
4.1.5 Showing Racket Examples	55
4.1.6 Multi-Page Sections	56
4.2 Style Guide	57
4.2.1 Prose and Terminology	57
4.2.2 Typesetting Code	58
4.2.3 Typesetting Prose	59
4.2.4 Section Titles	60
4.2.5 Indexing	60
4.2.6 Examples	60
4.3 Manual Forms	61
4.3.1 Typesetting Code	61
4.3.2 Documenting Modules	71
4.3.3 Documenting Forms, Functions, Structure Types, and Values	75
4.3.4 Documenting Classes and Interfaces	90
4.3.5 Documenting Signatures	92
4.3.6 Various String Forms	93
4.3.7 Links	94
4.3.8 Indexing	97
4.3.9 Images	98
4.3.10 Bibliography	98
4.3.11 Version History	99
4.3.12 Miscellaneous	99
4.3.13 Index-Entry Descriptions	101
4.3.14 Manual Rendering Style	103
4.4 Racket	104
4.5 Evaluation and Examples	108
4.6 In-Source Documentation	113
4.6.1 Source Annotations for Documentation	114
4.6.2 Extracting Documentation from Source	117
4.7 BNF Grammars	118
4.8 Compatibility Libraries	120
4.8.1 Compatibility Structures And Processing	120
4.8.2 Compatibility Basic Functions	127
5 Literate Programming	128
5.1 <code>scribble/lp</code> Language	129
5.2 <code>scribble/lp-include</code> Module	129

6 Low-Level Scribble API	131
6.1 Scribble Layers	131
6.1.1 Typical Composition	131
6.1.2 Layer Roadmap	133
6.2 @ Reader Internals	135
6.2.1 Using the @ Reader	135
6.2.2 Syntax Properties	135
6.2.3 Adding @-expressions to a Language	137
6.2.4 Interface	137
6.3 Structures And Processing	140
6.3.1 Parts, Flows, Blocks, and Paragraphs	141
6.3.2 Tags	144
6.3.3 Styles	145
6.3.4 Collected and Resolved Information	145
6.3.5 Structure Reference	146
6.3.6 HTML Style Properties	166
6.3.7 Latex Style Properties	170
6.4 Renderers	171
6.4.1 Rendering Driver	171
6.4.2 Base Renderer	172
6.4.3 Text Renderer	176
6.4.4 Markdown Renderer	176
6.4.5 HTML Renderer	177
6.4.6 Latex Renderer	178
6.4.7 PDF Renderer	178
6.4.8 Contract (Blue boxes) Renderer	178
6.5 Decoding Text	179
6.6 Document Language	184
6.6.1 <code>scribble/doclang</code>	185
6.7 Document Reader	186
6.8 Cross-Reference Utilities	186
6.9 Tag Utilities	190
6.10 Blue Boxes Utilities	192
6.11 Extending and Configuring Scribble Output	193
6.11.1 Implementing Styles	193
6.11.2 Configuring Output	195
6.11.3 Base CSS Style Classes	196
6.11.4 Manual CSS Style Classes	199
6.11.5 Base Latex Macros	200
6.11.6 Latex Prefix Support	202
7 Running scribble	203
7.1 Extra and Format-Specific Files	204
7.2 Handling Cross-References	204
7.3 Selecting an Image Format	205
7.4 Passing Command-Line Arguments to Documents	206

Index

207

Index

207

1 Getting Started

No matter what you want to do with Scribble, it's best to start by generating a few simple HTML and/or PDF documents. This chapter steps you through the basics, and it ends in §1.10 "Next Steps" with goal-specific advice on how to continue.

1.1 A First Example

Create a file "mouse.scrbl" with this content:

```
#lang scribble/base

@title{On the Cookie-Eating Habits of Mice}

If you give a mouse a cookie, he's going to ask for a
glass of milk.
```

The first line's `#lang scribble/base` indicates that the file implements a Scribble document. The document starts in "text mode," and the `@` character escapes to operators like `title`, where the curly braces return to text mode for the arguments to the operator. The rest is document content.

Now run the `scribble` command-line program, specifying a mode for the kind of document that you want as output:

- Run

```
scribble mouse.scrbl
```

to generate HTML as "mouse.html". You may notice that the apostrophe in "he's" turned into a curly apostrophe.

- Run

```
scribble --htmls mouse.scrbl
```

to generate HTML as "mouse/index.html". Sub-sections (which we add next) will appear as separate HTML files in the "mouse" directory.

- Run

```
scribble --pdf mouse.scrbl
```

to generate PDF as "mouse.pdf". This will work only if you have `pdflatex` installed. If you'd like to see the intermediate Latex, try

```
scribble --latex mouse.scrbl
```

to generate "mouse.tex".

See §7 “Running scribble” for more information on the scribble command-line tool.

1.2 Multiple Sections

Add more text to "mouse.scrbl" so that it looks like this:

```
#lang scribble/base

@title{On the Cookie-Eating Habits of Mice}

If you give a mouse a cookie, he's going to ask for a
glass of milk.

@section{The Consequences of Milk}

That ``squeak'' was the mouse asking for milk. Let's
suppose that you give him some in a big glass.

He's a small mouse. The glass is too big---way too
big. So, he'll probably ask you for a straw. You might as
well give it to him.

@section{Not the Last Straw}

For now, to handle the milk moustache, it's enough to give
him a napkin. But it doesn't end there... oh, no.
```

Now, after the first paragraph of the paper, we have two sub-sections, each created by calling `section` to generate a sub-section declaration. The first sub-section has two paragraphs. The second section, as initiated by the result of the second `section` call, has a single paragraph.

Run the `scribble` command(s) from §1.1 “A First Example” again. You may notice the curly double-quotes in the output, and the `---` turned into an em dash.

1.3 Splitting the Document Source

As a document grows larger, it’s better to split sections into separate source files. The `include-section` operation incorporates a document defined by a ".scrbl" file into a larger document.

To split the example document into multiple files, change "mouse.scrbl" to just

```

#lang scribble/base

@title{On the Cookie-Eating Habits of Mice}

If you give a mouse a cookie, he's going to ask for a
glass of milk.

@include-section["milk.scrbl"]
@include-section["straw.scrbl"]

```

Create "milk.scrbl" and "straw.scrbl" in the same directory as "mouse.scrbl". In "milk.scrbl", put

```

#lang scribble/base

@title{The Consequences of Milk}

That ``squeak'' was the mouse asking for milk...

```

and in "straw.scrbl", put

```

#lang scribble/base

@title{Not the Last Straw}

For now, to handle the milk moustache, ...

```

Notice that the new files both start with `#lang`, like the original document, and the `sections` from the original document become `titles` in the new documents. Both "milk.scrbl" and "straw.scrbl" are documents in their own right with their own titles, and they can be individually rendered using `scribble`. Running `scribble` on "mouse.scrbl", meanwhile, incorporates the smaller documents into one document that is the same as before.

1.4 Document Styles

Scribble currently supports only one form of HTML output. You can replace the "scribble.css" file for the generated pages, and that's about it. (We expect to add more styles in the future.)

For Latex-based PDF output, Scribble includes support for multiple page-layout configurations. The "mouse.scrbl" example so far uses the default Latex style. If you plan on submitting the paper to a workshop on programming languages, then—well, you probably need a different topic. But you can start making the current content look right by changing the first line to


```
#lang scribble/sigplan
```

If you're instead working toward Racket library documentation, try changing the first line to

```
#lang scribble/manual
```

which produces output with a separate title page, initial content on that page (intended as a brief orientation to the document), and top-level sections turned into chapters that each start on a new page. If you have split the document into multiple files, the first line of the main document file determines the output format.

Using `scribble/sigplan` or `scribble/manual` does not change the rendered HTML for a document—aside from `scribble/manual` adding a version number—but it changes the set of bindings available in the document body. For example, with `scribble/sigplan`, the introductory text can be marked as an abstract:

```
#lang scribble/sigplan

@title{On the Cookie-Eating Habits of Mice}

@abstract{If you give a mouse a cookie, he's going to
          ask for a glass of milk.}

@section{The Consequences of Milk}

....
```

When rendered as HTML, the abstract shows up as an inset paragraph. If you try to use `abstract` with the `scribble/base` or `scribble/manual` language, then you get an error, because `abstract` is not defined.

When a document is implemented across multiple files, changing the language of the main document can set the style for all of the parts, but it does not introduce bindings into the other part files. For example, if you change the language of "mouse.scrbl" to `scribble/sigplan`, then `abstract` becomes available in "mouse.scrbl" but not in "milk.scrbl" or "straw.scrbl". In other words, operator names are lexically scoped.

1.5 More Functions

The `scribble/base` language provides a collection of basic operations (and The `scribble/sigplan` and `scribble/manual` are supersets of `scribble/base`). Many of the operations are style variations that you can apply to text:

```
He's a @smaller{small mouse}. The glass is too
@larger{big}---@bold{way @larger{too @larger{big}}}. So, he'll
@italic{probably} ask you for a straw.
```

which renders as

He's a small mouse. The glass is too **big—way too big**. So, he'll *probably* ask you for a straw.

As you would expect, calls to functions like `smaller`, `larger`, and `bold` can be nested in other calls. They can also be nested within calls to `title` or `section`:

```
@section{@italic{Not} the Last Straw}
```

1.5.1 Centering

The `centered` operation centers a flow of text:

```
If a mouse eats all your cookies, put up a sign that says
@centered{
  @bold{Cookies Wanted}

  @italic{Chocolate chip preferred!}
}
and see if anyone brings you more.
```

which renders as

If a mouse eats all your cookies, put up a sign that says

Cookies Wanted

Chocolate chip preferred!

and see if anyone brings you more.

1.5.2 Margin Notes

The `margin-note` operation is used in a similar way, but the rendered text is moved to the margins.

If you use `margin-note`, then the content shows up over here.

1.5.3 Itemizations

The `itemlist` operation creates a sequence of bulleted text, where the `item` operation groups text to appear in a single bullet. The `itemlist` operation is different from the others that we have seen before, because it only accepts values produced by `item` instead of arbitrary text. This difference is reflected in the use of `[...]` for the arguments to `itemlist` instead of `{...}`:

```
@centered{@bold{Notice to Mice}}

@itemlist[@item{We have cookies for you.}
          @item{If you want to eat a cookie,
                you must bring your own straw.}]
```

which renders as

Notice to Mice

- We have cookies for you.
- If you want to eat a cookie, you must bring your own straw.

1.5.4 Tables

The `tabular` function takes a list of lists to organize into a two-dimensional table. By default, no spacing is added between columns, so supply a `#:sep` argument to acts as a column separator. For example,

```
@tabular[#:sep @hspace[1]
         (list (list @bold{Animal} @bold{Food})
              (list "mouse"      "cookie")
              (list "moose"      "muffin"))]
```

renders as

Animal	Food
mouse	cookie
moose	muffin

1.6 Text Mode vs. Racket Mode for Arguments

When `[...]` surrounds the arguments of an operation, the argument expressions are in Racket mode rather than text mode. Even in Racket mode, `@` can be used to apply operations; once

the `@` syntax is enabled through a language like `scribble/base` (as opposed to `racket`), it behaves the same in both Racket mode and text mode.

One advantage of using Racket mode for the arguments to `itemlist` is that we can pass a keyword-tagged optional argument to `itemlist`. In particular, if you want a list with numbers instead of bullets, supply the `'ordered` style to `itemlist` using the `#:style` keyword:

```
@itemlist[#:style 'ordered
  @item{Eat cookie.}
  @item{Drink milk.}
  @item{Wipe mouth.}
  @item{...}]
```

An operation doesn't care whether it's used with `[...]` or `{...}`. Roughly, `{...}` forms an argument that is a string. (Only roughly, though. Newlines or uses of `@` within `{...}` complicate the picture, and we'll get back to that soon.) So,

```
@italic{Yummy!}
```

is equivalent to

```
@italic["Yummy!"]
```

which is equivalent to the Racket expression

```
(italic "Yummy!")
```

These equivalences explain why Scribble functions are documented in Racket notation. If you're reading this in HTML format, you can click [italic](#) above to access its documentation. The documentation won't completely make sense, yet, but it will by the end of this chapter.

What if you want to provide arguments in text mode, but you also want to supply other optional arguments? You can use both `[...]` and `{...}` for an operation, as long as the `[...]` is first, and as long as no character separate the closing `]` from the opening `{`. For example, calling `italic` is the same as using `elem` with the `'italic` style:

```
@elem[#:style 'italic]{Yummy!}
```

You can also *omit* both `[...]` and `{...}`. In that case, the Racket expression after `@` is used directly instead of applied as an operation. For example,

```
1 plus 2 is @(number->string (+ 1 2)).
```

renders as

1 plus 2 is 3.

The call to `number->string` is needed because a naked number is not valid as document content.

1.7 @ Syntax Basics

The `@` notation provided by Scribble is just another way of writing Racket expressions. Scribble documents could be constructed using normal Racket notation, without using `@` at all, but that would be inconvenient for most purposes. The `@` notation makes dealing with textual content much easier.

Whether in text mode or Racket mode, `@` in a document provides an escape to Racket mode. The basic 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 normal Racket notation, while a `<text-body>` is itself in text mode. A `<cmd>` obviously must not start with `[` or `{`, even though Racket forms could otherwise start with those characters.

The expansion of just `@<cmd>` into Racket 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 Racket strings.

In practice, the `<cmd>` is normally a Racket 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 Racket arguments, instead. Even if an operation's argument is a string, if the string is not used as content text (but instead used as, say, a hyperlink label), then the string is typically provided through `[...]` instead of `{...}`. Sometimes, both `[...]` and `{...}` are used, where the former surround Racket arguments that precede text to typeset. Finally, if a form is a purely Racket-level form with not typeset result, such as a `require` to import more operations, then typically just `@` is used.

For example the text-mode stream

```
@(require scriblib/figure)

@section[:tag "poetry"]{Of Mice and Cookies}
See @secref["milk"].

@section[:tag "milk"]{@italic{Important} Milk Supplies}
@figure["straw" @elem{A straw}]{@image["straw.png"]}
```

is equivalent to the Racket-mode sequence

```
(require scriblib/figure) "\n"
"\n"
(section #:tag "poetry" "Of Mice and Cookies") "\n"
"See " (secref "milk") "." "\n"
"\n"
(section #:tag "milk" (italic "Important") " Milk Supplies") "\n"
(figure "straw" (elem "A straw") (image "straw.png")) "\n"
```

Besides showing how different argument conventions are used for different operations, the above example illustrates how whitespace is preserved in the Racket form of a text-mode stream—including newlines preserved as their own strings. Notice how the second `section` gets two arguments for its content, since the argument content for `section` in the source stream includes both the use of an operator and additional text. When an operation like `section` or `italic` accepts content to typeset, it normally accepts an arbitrary number of arguments that together form the content.

In addition to its role for command, a `@` can be followed by `;` to start a comment. If the character after `;` is `{`, then the comment runs until a matching `}`, otherwise the comment runs until the end-of-line:

```
@;{ <comment> }
@; <line-comment>
```

For more information on the syntax of `@`, see §2 “@ Syntax”. The full syntax includes a few more details, such as brackets like `||{...}||` for text-mode arguments while disabling `@` between the brackets.

1.8 Decoding Sequences

In a document that starts `#lang scribble/base`, the top level is a text-mode stream, just like the `<text-body>` in a `@` form. As illustrated in the previous section, such a top-level sequence corresponds to a mixture of Racket-mode strings and operation applications. There's an implicit operation, `decode`, that wraps the whole document to consume this mixture of strings and other values and turn them into a document description.

The `decode` operation implements *flow decoding*, which takes a document stream and breaks it up into sections and paragraphs. Blank lines delimit paragraphs, and the results of operations like `title` and `section` generate “here’s the title” or “a new section starts here” declarations that are recognized by `decode`.

A different but related *content decoding* takes place within a paragraph or section title. Content decoding is responsible for converting `---` to an em dash or for converting `"` and `'` to suitable curly quotes.

The decoding process for document’s stream is ultimately determined by the `#lang` line that starts the document. The `scribble/base`, `scribble/manual`, and `scribble/sigplan` languages all use the same `decode` operation. The `scribble/text` language, however, acts more like a plain-text generator and preprocessor, and it does not perform any such decoding rules. (For more on `scribble/text`, see *Scribble as Preprocessor*.)

When the flow decoder is used, after it breaks the input stream into paragraphs, it applies content decoding to strings within the paragraph. When content is wrapped with an operation, however, content decoding does not apply automatically. An operation is responsible for calling a content or flow decoder as it sees fit. Most operations call the decoder; for example, `italic`, `bold`, `smaller`, etc., all decode their arguments. Similarly, `title` and `section` decode the given content for the title or section name. The `literal` and `verbatim` operators, however, do not decode the given strings. For example,

```
@verbatim{---}
```

renders as

```
---
```

Don’t confuse decoding with the expansion of `@` notation. The source form

```
@verbatim{@(number->string (+ 1 2))}
```

renders as

```
3
```

More precisely, languages like `scribble/base` apply `decode` only after lifting out all definitions and imports from the document stream.

because the source is equivalent to

```
(verbatim (number->string (+ 1 2)))
```

where `(number->string (+ 1 2))` is evaluated to produce the argument to `verbatim`. The `{...}` style of brackets is often used with `verbatim`, because `{...}` disables `@` notation for arguments. For example,

```
@verbatim|{@(number->string (+ 1 2))}|
```

renders as

```
@(number->string (+ 1 2))
```


1.9 Pictures

Any value that is convertible to an image can be used directly within a Scribble document. Functions from the `pict` and `2htdp/image` libraries, for example, generate images. For example,

```
@(require pict)
```

```
This cookie has lost its chocolate chips:  
@(colorize (filled-ellipse 40 40) "beige").
```

renders as

This cookie has lost its chocolate chips:  .

1.10 Next Steps

If your immediate goal is to document a Racket library or write literate programs, skip to §4.1 “Getting Started with Documentation”, and then go back to §2 “@ Syntax” and other chapters.

If you are more interested in producing documents unrelated to Racket, continue with §2 “@ Syntax” and then §3 “High-Level Scribble API”. Move on to §6 “Low-Level Scribble API” when you need more power.

If you are interested in text generation and preprocessing, continue with §2 “@ Syntax”, but then switch to *Scribble as Preprocessor*.

2 @ Syntax

The Scribble @ notation is designed to be a convenient facility for free-form text in Racket code, where “@” was chosen as one of the least-used characters in existing Racket code. An @-expression is simply an S-expression in disguise.

Typically, @ notation is enabled through `scribble/base` or similar languages, but you can also add @ notation to an S-expression-based language using the `at-exp` meta-language. For example,

```
#lang at-exp racket
(define v '@op{str})
```

is equivalent to

```
#lang racket
(define v '(op "str"))
```

Using `#lang at-exp racket` is probably the easiest way to try the examples in this chapter.

2.1 The Scribble Syntax at a Glance

To review §1.7 “@ Syntax Basics”, the concrete syntax of @-forms is roughly

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

where all three parts after @ are optional, but at least one should be present. (Spaces are not allowed between the three parts.) 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 Racket code that the input is translated into. The common case is when *<cmd>* is a Racket identifier, which reads as a plain Racket form, with datum arguments and/or string arguments.

Here is one example:

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

The example shows how an input syntax is read as Racket syntax, not what it evaluates to. If you want to see the translation of an example into S-expression form, add a quote in front of it in a `#lang at-exp racket` module. For example, running

```
#lang at-exp racket
'@foo{blah blah blah}
```

in DrRacket prints the output

```
(foo "blah blah blah")
```

while omitting the quote

```
#lang at-exp racket
@foo{blah blah blah}
```

triggers a syntax error because `foo` is not bound, and

```
#lang at-exp racket
(define (foo str) (printf "He wrote ~s.\n" str))
@foo{blah blah blah}
```

prints the output

```
He wrote "blah blah blah".
```

Here are more examples of @-forms:

```
@foo{blah "blah" (`blah'?)}
```

 reads as

```
(foo "blah \"blah\" (`blah'?"))
```

```
@foo[1 2]{3 4}
```

 reads as

```
(foo 1 2 "3 4")
```

```
@foo[1 2 3 4]
```

 reads as

```
(foo 1 2 3 4)
```

```
@foo[:width 2]{blah blah}
```

 reads as

```
(foo #:width 2 "blah blah")
```

```
@foo{blah blah
      yada yada}
```

 reads as

```
(foo "blah blah" "\n"
      "yada yada")
```

```
@foo{
  blah blah
  yada yada
}
```

 reads as

```
(foo
  "blah blah" "\n"
  "yada yada")
```

As seen in the last example, multiple lines and the newlines that separate them are parsed to multiple Racket strings. More generally, a *text-body* is made of text, newlines, and nested @-forms, where the syntax for @-forms is the same whether it's in a *text-body* context as in a Racket context. A *text-body* that isn't an @-form is converted to a string expression for its *parsed-body*; newlines and following indentations are converted to "`\n`" and all-space string expressions.

```

@foo{bar @baz{3}
      blah}      reads as  (foo "bar " (baz "3") "\n"
                        "blah")

@foo{@b{@u{3} @u{4}}
      blah}      reads as  (foo (b (u 3) " " (u "4")) "\n"
                        "blah")

@C{while (*(p++))
    *p = '\n';}  reads as  (C "while (*(p++))" "\n" " "
                        "*p = '\n';")

```

The command part of an @-form is optional as well. In that case, the @-form is read as a list, which usually counts as a function application, but it also useful when quoted with the usual Racket quote:

```

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

@{blah @[3]}    reads as  ("blah " (3))

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

```

Finally, we can also drop the datum and text parts, which leaves us with only the command— which is read as is, not within a parenthesized form. This is not useful when reading Racket code, but it can be used inside a text block to escape a Racket identifier. A vertical bar (|) can be used to delimit the escaped identifier when needed.

```

@foo           reads as  foo
@{blah @foo blah} reads as ("blah " foo " blah")
@{blah @foo: blah} reads as ("blah " foo: " blah")
@{blah @|foo|: blah} reads as ("blah " foo ": blah")

```

Actually, the command part can be any Racket expression (that does not start with [, {, or |), which is particularly useful with such escapes since they can be used with any expression.

```

@foo{(+ 1 2) -> @(+ 1 2)!} reads as (foo "(+ 1 2) -> " (+ 1 2) "!")
@foo{A @"string" escape}  reads as (foo "A string escape")

```

Note that an escaped Racket string is merged with the surrounding text as a special case. This is useful if you want to use the special characters in your string, but escaping braces are not necessary if they are balanced.

```

@foo{eli@"barzilay.org"} reads as (foo "eli@barzilay.org")

@foo{A @"{" begins a block"} reads as (foo "A { begins a block")

```

```
@C{while (*(p++)) {
  *p = '\n';
}}
reads as (C "while (*(p++)) {" "\n" " "
          "*p = '\n';" "\n"
          "}")
```

In some cases, a text contains many literal @s, which can be cumbersome to quote individually. For such case, braces have an alternative syntax: A block of text can begin with a “{|” and terminated accordingly with a “}|”. Furthermore, any nested @-forms must begin with a “|@”.

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

In cases when even this is not convenient enough, punctuation characters can be added between the | and the braces and the @ in nested forms. (The punctuation is mirrored for parentheses and <>s.) With this extension, Scribble syntax can be used as a “here string” replacement.

```
@foo|--{bar}@|{baz}--| reads as (foo "bar}@|{baz}")
@foo|<<{bar}@|{baz}>>| reads as (foo "bar}@|{baz}")
```

On the flip side of this is, how can an @ sign be used in Racket code? This is almost never an issue, because Racket strings and characters are still read the same, and @ is set as a non-terminating reader macro so it can be used in Racket identifiers anywhere except in the first character of an identifier. When @ must appear as the first character of an identifier, you must quote the identifier just like other non-standard characters in normal S-expression syntax: with a backslash or with vertical bars.

```
(define \@email "foo@bar.com") reads as (define @email "foo@bar.com")
(define |@atchar| #\@) reads as (define @atchar #\@)
```

Note that spaces are not allowed before a | or a {, or they will be part of the following text (or Racket 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}")
```

Finally, remember that the Scribble is just an alternate for S-expressions. Identifiers still get their meaning, as in any Racket code, through the lexical context in which they appear. Specifically, when the above @-form appears in a Racket expression context, the lexical environment must provide bindings for foo as a procedure or a macro; it can be defined, required, or bound locally (with let, for example).

```
> (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*."
```

2.2 The Command Part

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

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

When writing Racket 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* Racket 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@;    reads as  (foo "bar bazblah")
  blah}
```

Tip: if you use an editor in some Scheme mode without support for @-forms, balanced comments can be confusing, since the open brace looks commented out, and the closing one isn't. In such cases it is useful to “comment” out the closing brace too:

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

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

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 Racket 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")
```

2.3 The Datum Part

The datum part can contains arbitrary Racket 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 Racket forms that expect keyword-value arguments that precede the body of text arguments.

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

2.4 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}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 Racket 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 Racket expression, this makes `@` a general escape to arbitrary Racket 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\")")
```

2.4.1 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
      |@bar{is}
      Life!}| reads as (foo "Maze" "\n"
                       (bar "is") "\n"
                       "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 Racket 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.

2.4.2 Racket Expression Escapes

In some cases, you may want to use a Racket identifier (or a number or a boolean etc.) in a position that touches the following text; in these situations you should surround the escaped Racket expression by a pair of `|` characters. The text inside the bars is parsed as a Racket 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 Racket 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@|*
      *|y} reads as (foo "x" *
                    * "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@|
      |Carol} reads as (foo "Alice" "Bob"
                        "Carol")
```

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

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

2.4.3 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
                  newline here;}
      Second line} reads as (foo "First line"
                              "\n"
                              "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 @;
      single-@; reads as (foo "A long single-string arg.")
      string arg.}
```

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

2.4.4 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
  baz      reads as (foo
                  "bar" "\n"
                  "\n"
                  "baz")
}
@foo{
}          reads as (foo "\n")
@foo{
}          reads as (foo "\n"
                  "\n")
```

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

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 Racket REPL, you should be aware that the reader does not know about the “> ” prompt.)

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

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

```
@foo{
  a        reads as (foo " "
  b        "a" "\n" " "
  c}      "b" "\n"
          "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
  baz
  bbb}
reads as
(foo " bar" "\n"
 "baz" "\n" " "
 "bbb")

@text{Some @b{bold
text}, and
more text.}
reads as
(text "Some " (b "bold" "\n"
 "text"), and "\n"
 "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 Racket 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{                               (foo
  @|| bar @|| reads as           " bar " "\n"
  @|| baz}                          " baz")
```

3 High-Level Scribble API

3.1 Base Document Format

```
#lang scribble/base      package: scribble-lib
```

The `scribble/base` language provides functions and forms that can be used from code written either in Racket or with `@` expressions. It essentially extends `racket/base`, except that top-level forms within a module using the `scribble/base` language are treated as document content (like `scribble/doclang`).

The `scribble/base` name can also be used as a library with `require`, in which case it provides only the bindings defined in this section, and it also does not set the reader or set the default rendering format to the Racket manual format.

Functions provided by this library, such as `title` and `italic`, might be called from Racket as

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

They can also be called with `@` notation as

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

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

3.1.1 Document Structure

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

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.

The `style` argument can be a style structure, or it can be one of the following: a `#f` that corresponds to a “plain” style, a string that is used as a style name, a symbol that is used as a style property, or a list of symbols to be used as style properties. For information on styles, see `part`. For example, a style of `'toc` causes sub-sections to be generated as separate pages in multi-page HTML output.

The `tag-prefix` argument is propagated to the generated structure (see §6.3.2 “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. Use `"` as `vers` to suppress version rendering in the output.

The `date` argument is propagated to the `title-decl` structure via a `document-date` style property. Use `"` as `date` to suppress date rendering in Latex output.

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

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

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 #f string? (listof string?)) = #f
tag-prefix : (or/c #f string? module-path?) = #f
style : (or/c style? #f string? symbol? (listof symbol?)) = #f
pre-content : pre-content?

```

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 #f string? (listof string?)) = #f
tag-prefix : (or/c #f string? module-path?) = #f
style : (or/c style? #f string? symbol? (listof symbol?)) = #f
pre-content : pre-content?

```

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).

```

(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.

```

(author auth ...) → block?
  auth : content?

```

Generates a `paragraph` with style name `'author` to show the author(s) of a document, where each author is represented by content. Normally, this function is used after `title` for the beginning of a document. See also `author+email`.

```

(author+email author-name
              email
              [#:obfuscate? obfuscate?]) → element?
author-name : content?
email : string?
obfuscate? : any/c = #f

```

Combines an author name with an e-mail address. If `obfuscate?` is true, then the result obscures the e-mail address slightly to avoid address-harvesting robots.

Note that `author+email` is not a replacement for `author`. The `author+email` function is often used in combination with `author`.

3.1.2 Blocks

```
(para [#:style style] pre-content ...) → paragraph?  
  style : (or/c style? string? symbol? #f) = #f  
  pre-content : pre-content?
```

Creates a paragraph containing the decoded *pre-content* (i.e., parsed with `decode-paragraph`).

The *style* argument can be a style, `#f` to indicate a “plain” style, a string that is used as a style name, or a symbol that is used as a style name. (Note that `section` and `para` treat symbols differently as *style* arguments.)

```
(nested [#:style style] pre-flow ...) → nested-flow?  
  style : (or/c style? string? symbol? #f) = #f  
  pre-flow : pre-flow?
```

Creates a nested flow containing the decoded *pre-flow* (i.e., parsed with `decode-flow`).

The *style* argument is handled the same as `para`. The `'inset` and `'code-inset` styles cause the nested flow to be inset compared to surrounding text, with the latter particularly intended for inseting code.

```
(centered pre-flow ...) → nested-flow?  
  pre-flow : pre-flow?
```

Produces a nested flow whose content is centered.

```
(margin-note pre-flow ... [#:left? left?]) → block?  
  pre-flow : pre-flow?  
  left? : any/c = #f
```

Produces a nested flow that is typeset in the margin, instead of inlined.

If *left?* is true, then the note is shown on the opposite as it would normally be shown (which is the left-hand side for HTML output). Beware of colliding with output for a table of contents.

```
(margin-note* pre-content ... [#:left? left?]) → element?  
  pre-content : pre-content?  
  left? : any/c = #f
```

Produces an `element` that is typeset in the margin, instead of inlined. Unlike `margin-note`, `margin-note*` can be used in the middle of a paragraph; at the same time, its content is constrained to form a single paragraph in the margin.

```
(itemlist itm ... [#:style style]) → itemization?
  itm : items/c
  style : (or/c style? string? symbol? #f) = #f
```

Constructs an `itemization` given a sequence of items. Typical each `itm` is constructed by `item`, but an `itm` can be a block that is coerced to an `item`. Finally, `itm` can be a list or `splice` whose elements are spliced (recursively, if necessary) into the `itemlist` sequence.

The `style` argument is handled the same as `para`. The `'ordered` style numbers items, instead of just using a bullet.

```
items/c : flat-contract?
```

A contract that is equivalent to the following recursive specification:

```
(or/c item? block? (listof items/c) (spliceof items/c))
```

```
(item pre-flow ...) → item?
  pre-flow : pre-flow?
```

Creates an item for use with `itemlist`. 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.

```
(tabular cells
  [#:style style
   #:sep sep
   #:column-properties column-properties
   #:row-properties row-properties
   #:cell-properties cell-properties]) → table?
  cells : (listof (listof (or/c block? content? 'cont)))
  style : (or/c style? string? symbol? #f) = #f
  sep : (or/c block? content? #f) = #f
  column-properties : (listof any/c) = '()
  row-properties : (listof any/c) = '()
  cell-properties : (listof (listof any/c)) = '()
```

Creates a table with the given `cells` content, which is supplied as a list of rows, where each row has a list of cells. The length of all rows must match.

Use `'cont` in `cells` as a cell to continue the content of the preceding cell in a row in the space that would otherwise be used for a new cell. A `'cont` must not appear as the first cell in a row.

The `style` argument is handled the same as `para`.

If `sep` is not `#f`, it is inserted as a new column between every column in the table; note that any `table-columns` or `table-cells` property in `style` must take the added columns into account. Otherwise, the default style places no space between table columns.

The `column-properties`, `row-properties`, and `cell-properties` arguments specify style properties for the columns and cells of a table; see `table-columns` and `table-cells` for a description of recognized properties. The lists do not contain entries for columns potentially introduced for `sep`, and when non-empty, they are extended as needed to match the table size determined by `cells`:

- If the length of `column-properties` is less than the length of each row in `cells`, the last item of the list is duplicated to make the list long enough.
- If the length of `row-properties` is less than the length of `cells`, the last item of the list is duplicated to make the list long enough.
- If the length of `cell-properties` is less than the number of rows in `cells`, then the last element is duplicated to make the list long enough. Each list within `cell-properties` is treated like a `column-properties` list—expanded as needed to match the number of columns in each row.

Each element of `column-properties` or `row-properties` is either a list of style property values or a non-list element that is wrapped as a list. Similarly, for each list that is an element of `cell-properties`, the list's non-list elements are wrapped as nested lists.

If `column-properties` is non-empty, then its list of property lists is converted into a `table-columns` style property that is added to the style specified by `style`—or merged with an existing `table-columns` style property that matches the column shape of `cells`. In addition, if either `row-properties` or `cell-properties` is non-empty, the property lists of `column-properties` are merged with the property lists of `row-properties` and `cell-properties`. If `row-properties` or `cell-properties` is non-empty, the merged lists are converted into a `table-cells` style property that is added to the style specified by `style`—or merged with an existing `table-cells` style property that matches the shape of `cells`.

Changed in version 1.1 of package `scribble-lib`: Added the `#:column-properties`, `#:row-properties`, and `#:cell-properties` arguments.

Examples:

If the style lists for `column-properties` are both merged with `cell-properties` and converted to `table-columns`, then `style` will contain some redundant information. In that case, `column-attributes` properties will be used from `table-columns`, while other properties will be used from the merger into `table-cells`.

```

@tabular[#:sep @hspace[1]
  (list (list "soup" "gazpacho")
        (list "soup" "tonjiru"))]

@tabular[#:style 'boxed
  #:column-properties '(left right)
  #:row-properties '(bottom-border ())
  (list (list @bold{recipe} @bold{vegetable})
        (list "caldo verde" "kale")
        (list "kinpira gobō" "burdock")
        (list "makizushi" 'cont))]

```

Renders like:

```

soup gazpacho
soup tonjiru
recipe    vegetable
caldo verde    kale
kinpira gobō  burdock
makizushi

```

```

(verbatim [#:indent indent] elem ...+) → block?
  indent : exact-nonnegative-integer? = 0
  elem : content?

```

Typesets string *elems* in typewriter font with linebreaks specified by newline characters in string *elems*. Consecutive spaces in the string *elems* are converted to `hspace` to ensure that they are all preserved in the output. Additional space (via `hspace`) as specified by *indent* is added to the beginning of each line. A non-string *elem* is treated as content within a single line.

The string *elems* are *not* decoded with `decode-content`, so `(verbatim "---")` renders with three hyphens instead of an em dash. Beware, however, that *reading* `@verbatim` converts `@` syntax within the argument, and such reading occurs well before arguments to `verbatim` are delivered at run-time. To disable simple `@` notation within the `verbatim` argument, `verbatim` is typically used with `{...}` or similar brackets, like this:

```

@verbatim|{
  Use @bold{---} like this...
}|

```

which renders as

```

  Use @bold{---} like this...

```

while

```
@verbatim|{
  Use |@bold{---} like this...
}|
```

renders as

```
Use — like this...
```

Even with brackets like `|{...}|`, beware that consistent leading whitespace is removed by the parser; see §2.4.1 “Alternative Body Syntax” for more information.

See also [literal](#).

3.1.3 Text Styles and Content

```
(elem pre-content ... [:#:style style]) → element?
pre-content : pre-content?
style : (or/c style? string? symbol? #f) = #f
```

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

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

Like `elem`, but with style `'italic`.

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

Like `elem`, but with style `'bold`.

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

Similar to `elem`, but the `'tt` style is used for immediate strings and symbols among the *pre-content* arguments.

To apply the `'tt` style uniformly to all *pre-content* arguments, use `(elem #:style 'tt pre-content ...)`, instead.

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

Like `elem`, but with style `'subscript`.

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

Like `elem`, but with style `'superscript`.

```
(smaller pre-content ...) → element?  
  pre-content : pre-content?
```

Like `elem`, but with style `'smaller`. When uses of `smaller` are nested, text gets progressively smaller.

```
(larger pre-content ...) → element?  
  pre-content : pre-content?
```

Like `elem`, but with style `'larger`. When uses of `larger` are nested, text gets progressively larger.

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

The same as `italic`.

```
(literal str ...+) → element?  
  str : string?
```

Produces an element containing literally `str`s with no decoding via `decode-content`.

Beware that `@` for a `literal` call performs some processing before delivering arguments to `literal`. The `literal` form can be used with `!{...}` or similar brackets to disable `@` notation within the `literal` argument, like this:

```
@literal|{@bold{---}}|
```

which renders as

```
@bold{---}
```

See also `verbatim`.

```
(image path  
  [#:scale scale  
  #:suffixes suffixes  
  #:style style]  
  pre-content ...) → image-element?
```

```

path : (or/c path-string? (cons/c 'collects (listof bytes?)))
scale : real? = 1.0
suffixes : (listof #rx"^[.]") = null
style : (or/c style? string? symbol? #f) = #f
pre-content : pre-content?

```

Creates an image element from the given path. The decoded *pre-content* serves as the alternate text for contexts where the image cannot be displayed.

If *path* is a relative path, it is relative to the current directory, which is set by `raco setup` to the directory of the main document file. (In general, however, it's more reliable to express relative paths using `define-runtime-path`.) Instead of a path or string, the *path* argument can be a result of `path->main-collects-relative`.

The *scale* argument sets the images scale relative to its default size as determined by the content of *path*. For HTML output, the resulting *image-element* is rendered with an `img` or `object` (for SVG) tag, and *scale* adjusts the `width` and `height` attributes; a class name or other attributes in *style* can effectively override that size.

The strings in *suffixes* are filtered to those supported by given renderer, and then the acceptable suffixes are tried in order. The HTML renderer supports `".png"`, `".gif"`, and `".svg"`, while the Latex renderer supports `".png"`, `".pdf"`, and `".ps"` (but `".ps"` works only when converting Latex output to DVI, and `".png"` and `".pdf"` work only for converting Latex output to PDF).

Note that when the *suffixes* list is non-empty, then the *path* argument should not have a suffix.

Changed in version 1.3 of package `scribble-lib`: Added the `#:style` argument.

3.1.4 Spacing

```
(linebreak) → element?
```

Produces an element that forces a line break.

```
(nonbreaking pre-content ...) → element?
pre-content : pre-content?
```

Like `elem`, but line breaks are suppressed while rendering the content.

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

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

`~` : `string?`

A string containing the non-breaking space character, which is equivalent to `'nbsp` as an element.

`-~-` : `string?`

A string containing the non-breaking hyphen character.

`?-` : `string?`

A string containing the soft-hyphen character (i.e., a suggestion of where to hyphenate a word to break it across lines when rendering).

`._` : `element?`

Generates a period that ends an abbreviation in the middle of a sentence, as opposed to a period that ends a sentence (since the latter may be typeset with extra space). Use `@._` in a document instead of just `.` for an abbreviation-ending period that is preceded by a lowercase letter and followed by a space.

See `.__` for an example.

`.__` : `element?`

Generates a period that ends a sentence (which may be typeset with extra space), as opposed to a period that ends an abbreviation in the middle of a sentence. Use `@.__` in a document instead of just `.` for a sentence-ending period that is preceded by an uppercase letter.

The following example illustrates both `._` and `.__`:

```
#lang scribble/base
My name is Mr@._ T@.__ I pity the fool who can't typeset punctuation.
```

3.1.5 Links

```
(hyperlink url
  pre-content ...
  [#:underline? underline?
   #:style style]) → element?
url : string?
pre-content : pre-content?
underline? : any/c = #t
style : (or/c style? string? symbol? #f)
       = (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.

```
(url dest) → element?  
  dest : string?
```

Generates a literal hyperlinked URL.

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

Inserts a reference to the section tagged *tag*.

If *#:doc 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 Racket reference, *module-path* would be `'(lib "scribblings/reference/reference.scrbl")`.

The *#:tag-prefixes prefixes* argument similarly supports selecting a particular section as determined by a path of tag prefixes. When a *#:doc* argument is provided, then *prefixes* should trace a path of tag-prefixed subsections to reach the *tag* section. When *#:doc* is not provided, the *prefixes* path is relative to any enclosing section (i.e., the youngest ancestor that produces a match).

For HTML output, the generated reference is the hyperlinked title of the elements in the section's title content, except that elements with the `'aux` style property are omitted in the hyperlink label. If *underline?* is *#f*, then the hyperlink is rendered in HTML without an underline.

For Latex output, the generated reference's format depends on the document style. By default, only the section number is shown in the reference, but the `scribble/manual` style shows the title after the section number. Customize the output (see §6.11 “Extending and Configuring Scribble Output”) by redefining the `\BookRef`, etc., macros (see §6.11.5 “Base Latex Macros”).

In Racket documentation that is rendered to HTML, clicking on a section title normally shows the `secref` call that is needed to link to the section.

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

Like `secref`, but if the rendered form of the reference starts with a word (e.g., “section”), then the word is capitalized.

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

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

In addition to `secref`’s arguments, `seclink` supports a `indirect?` argument. When `indirect?` is true, then the section hyperlink’s resolution in HTML is potentially delayed; see `'indirect-link` for `link-element`.

```
(other-doc module-path
  [#:underline? underline?
   #:indirect indirect]) → element?
module-path : module-path?
underline? : any/c = #t
indirect : (or/c #f content?) = #f
```

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.

If `indirect` is not `#f`, then the link’s resolution in HTML can be delayed, like `seclink` with `#:indirect? #t`. The `indirect` content is prefixed with “the” and suffixed with “documentation” to generate the rendered text of the link.

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

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

```
(elemref t
  pre-content ...
  [#:underline? underline?]) → element?
  t : (or/c tag? string?)
  pre-content : pre-content?
  underline? : any/c = #t
```

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

3.1.6 Indexing

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

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.

Use `index` when an index entry should point to a specific word or phrase within the typeset document (i.e., the *pre-content*). Use `section-index`, instead, to create an index entry that leads to a section, instead of a specific word or phrase within the section.

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

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 : pre-content?
```

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 within the index.

```
(index-section [#:tag tag]) → part?  
  tag : (or/c #f 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.

3.1.7 Tables of Contents

```
(table-of-contents) → delayed-block?
```

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 [#:style style]) → delayed-block?  
  style : (or/c symbol? #f) = #f
```

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.

The meaning of the `style` argument depends on the output type, but `'immediate-only` normally creates a table of contents that contains only immediate sub-sections of the enclosing section. See also the `'quiet` style of `part` (i.e., in a `part` structure, not supplied as the `style` argument to `local-table-of-contents`), which normally suppresses sub-part entries in a table of contents.

3.1.8 Tags

The exports of `scribble/tag` are all re-exported by `scribble/base`.

3.2 Racket Manual Format

The `scribble/manual` language is a major component of Scribble, and it is documented in its own chapter: §4 “Scribbling Documentation”.

3.3 Book Format

```
#lang scribble/book      package: at-exp-lib
```

The `scribble/book` language is like `scribble/base`, but configured with Latex style defaults to use the standard book class. Top-level sections are rendered as Latex chapters.

3.4 Report Format

```
#lang scribble/report     package: at-exp-lib
```

The `scribble/report` language is like `scribble/book`, but configured with Latex style defaults to use the standard report class.

3.5 SIGPLAN Paper Format

```
#lang scribble/sigplan    package: scribble-lib
```

The `scribble/sigplan` language is like `scribble/base`, but configured with Latex style defaults to use the "sigplanconf.cls" class file that is included with Scribble.

| `preprint`

Enables the `preprint` option. Use `preprint` only on the same line as `#lang`, with only whitespace (or other options) between `scribble/sigplan` and `preprint`:

```
#lang scribble/sigplan @preprint
```

| `10pt`

Enables the `10pt` option. Use `10pt` only on the same line as `#lang`, with only whitespace (or other options) between `scribble/sigplan` and `10pt`:

```
#lang scribble/sigplan @10pt
```

| `nocopyright`

Enables the `nocopyright` option. Use `nocopyright` only on the same line as `#lang`, with only whitespace (or other options) between `scribble/sigplan` and `nocopyright`:

```
#lang scribble/sigplan @nocopyright
```

| `onecolumn`

Enables the `onecolumn` option. Use `onecolumn` only on the same line as `#lang`, with only whitespace (or other options) between `scribble/sigplan` and `onecolumn`:

```
#lang scribble/sigplan @onecolumn
```

| `notimes`

Disables the use of `\usepackage{times}` in the generated LaTeX output. Use `onecolumn` only on the same line as `#lang`, with only whitespace (or other options) between `scribble/sigplan` and `notimes`:

```
#lang scribble/sigplan @notimes
```

| `noqcourier`

Disables the use of `\usepackage{qcourier}` in the generated LaTeX output. Use `onecolumn` only on the same line as `#lang`, with only whitespace (or other options) between `scribble/sigplan` and `noqcourier`:

```
#lang scribble/sigplan @noqcourier
```

The `10pt`, `preprint`, `nocopyright`, `onecolumn`, `notimes`, and `noqcourier` options can be used together and may appear in any order.

```
(abstract pre-content ...) → block?  
pre-content : pre-content?
```

Generates a nested flow for a paper abstract.

```
(include-abstract module-path)
```

Similar to `include-section`, but incorporates the document in the specified module as an abstract. The document must have no title or sub-parts.

```
(subtitle pre-content ...) → element?  
pre-content : pre-content?
```

Use as the last argument to `title` to specify a subtitle.

```
(authorinfo name affiliation email) → block?  
name : pre-content?  
affiliation : pre-content?  
email : pre-content?
```

A replacement for `author` that associates an affiliation and e-mail address with the author name.

```
(conferenceinfo conference location) → block?
  conference : pre-content?
  location : pre-content?
(copyrightyear content ...) → block?
  content : pre-content?
(copyrightdata content ...) → block?
  content : pre-content?
(doi content ...) → block?
  content : pre-content?
(exclusive-license) → block?
```

Declares information that is collected into the copyright region of the paper.

```
(category CR-number
  subcategory
  third-level
  [fourth-level]) → content?
  CR-number : pre-content?
  subcategory : pre-content?
  third-level : pre-content?
  fourth-level : (or/c #f pre-content?) = #f
(terms content ...) → content?
  content : pre-content?
(keywords content ...) → content?
  content : pre-content?
```

Typesets category, term, and keyword information for the paper, which is normally placed immediately after an `abstract` form. See also <http://www.acm.org/about/class/how-to-use>.

For `category`, the `subcategory` argument should be in titlecase (i.e., capitalize the first letter of each word) and a phrase at the level of “Programming Languages” or “Software Engineering” (as opposed to a category like “Software” or a third-level name like “Concurrent Programming” or “Processors”). A `third-level` phrase should be in titlecase. A `fourth-level` phrase, if any, should not be capitalized.

For `terms`, each general term should be in titlecase. Terms are usually drawn from a fixed list, and they are usually optional.

For `keywords`, capitalize only the first letter of the first word, separate phrases by commas, and do not include “and” before the last one. Keywords should be noun phrases, not adjectives.

3.6 JFP Paper Format

```
#lang scribble/jfp      package: scribble-lib
```

The `scribble/jfp` language is like `scribble/base`, but configured with Latex style defaults to use the "jfp1.cls" class file. The class file is not included with Scribble due to license issues, but if the file is not manually installed into the `scribble/jfp` collections, then it is downloaded on demand to (`find-system-path 'addon-dir`).

Latex output with `scribble/jfp` uses a main-document version supplied to `title` as the short-form document name (to be used in page headers).

```
(abstract pre-content ...) → block?  
  pre-content : pre-content?
```

Generates a nested flow for a paper abstract.

```
(include-abstract module-path)
```

Similar to `include-section`, but incorporates the document in the specified module as an abstract. The document must have no title or sub-parts.

```
(author name ...) → block?  
  name : pre-content?
```

A replacement for `author` from `scribble/base`.

```
((author/short short-name ...) long-name ...) → block?  
  short-name : pre-content?  
  long-name : pre-content?
```

Like `author`, but allows the short-form names (to be used in page headers) to be specified separately from the long-form name.

```
(affiliation place ...) → element?  
  place : pre-content?  
(affiliation-mark mark ...) → element?  
  mark : pre-content?  
(affiliation-sep) → element?
```

Use `affiliation` within `author` or the long-name part of `author/short` to specify affiliations after all authors. If different authors have different affiliations, use `affiliation-mark` with a number after each author, and then use `affiliation-mark` before each different affiliation within a single `affiliation`, using (`affiliation-sep`) to separate affiliations.

3.7 LNCS Paper Format

```
#lang scribble/lncs      package: at-exp-lib
```

The `scribble/lncs` language is like `scribble/base`, but configured with Latex style defaults to use the "lncs.cls" class file. The class file is not included with Scribble due to license issues, but if the file is not manually installed into the `scribble/lncs` collection, then it is downloaded on demand to (`find-system-path 'addon-dir`).

```
(abstract pre-content ...) → block?  
pre-content : pre-content?
```

Generates a nested flow for a paper abstract.

```
(include-abstract module-path)
```

Similar to `include-section`, but incorporates the document in the specified module as an abstract. The document must have no title or sub-parts.

```
(authors auth ...)  
  
auth = (author pre-content-expr ...)  
       | (author #:inst str-expr pre-content-expr ...)  
  
pre-content-expr : pre-content?  
str-expr : string?
```

A replacement for `author` from `scribble/base`.

The `#:inst` should be a number that matches up to one of the arguments to `institutes`.

```
author
```

For use only in authors.

```
(institutes (institute pre-content-expr ...) ...)  
  
pre-content-expr : pre-content?
```

The `pre-content-exprs` are used as the institutions of the authors.

```
institute
```

For use only in institutes.

```
(email pre-content-expr ...)
```

Specifies an email address; must be used inside `institute`.

3.7.1 Example

Here is an example of a paper written in the LNCS format:

```
#lang scribble/lncs

@authors[@author[#:inst "1"]{Lauritz Darragh}
          @author[#:inst "2"]{Nikolaj Kyran}
          @author[#:inst "2"]{Kirsten Gormlaith}
          @author[#:inst "2"]{Tamaz Adrian}]

@institutes[
  @institute["University of Southeast Boston"
             @linebreak[]
             @email|{darragh@cs.seboston.edu}||]
  @institute["University of Albion"
             @linebreak[]
             @email|{{nkyran,gorm,tamaz}@cs.albion.ac.uk}||]

@title{Arak: Low-Energy, Interposable Theory}

@abstract{The implications of client-server symmetries have been
far-reaching and pervasive. Given the current status of
constant-time theory, mathematicians daringly desire the synthesis
of rasterization, which embodies the essential principles of
algorithms. In this work, we describe a client-server tool for
investigating flip-flop gates (Arak), verifying that the
producer-consumer problem can be made homogeneous, secure, and
wireless.}
```

For more randomly
generated papers,
see SCIGen: <http://pdos.csail.mit.edu/scigen>

4 Scribbling Documentation

The `scribble/manual` language and associated libraries provide extensive support for documenting Racket libraries. The most significant aspect of support for documentation is the way that source-code bindings are connected to documentation sites through the module namespace—a connection that is facilitated by the fact that Scribble documents are themselves modules that reside in the same namespace. §4.1 “Getting Started with Documentation” provides an introduction to using Scribble for documentation, and the remaining sections document the relevant libraries and APIs in detail.

4.1 Getting Started with Documentation

Although the `scribble` command-line utility generates output from a Scribble document, documentation of Racket libraries is normally built by `raco setup`. This chapter emphasizes the `raco setup` approach, which more automatically supports links across documents.

See §1 “Getting Started” for information on using the `scribble` command-line utility.

4.1.1 Setting Up Library Documentation

To document a collection, including a collection implemented by a package:

- Create a file in your collection with the file extension `".scrbl"`. Beware that the file name you choose will determine the output directory’s name, and the directory name must be unique across all installed documents. The remainder of these instructions assume that the file is called `"manual.scrbl"` (but pick a more specific name in practice).
- Start `"manual.scrbl"` like this:

```
#lang scribble/manual
```

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

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

The first line starts the file in “text” mode and selects the Racket manual output format. It also introduces bindings like `title` and `racket` for writing Racket documentation.

- Add the following entry to your collection’s `"info.rkt"`:

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

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.rkt" module, here's a suitable complete module:

```
#lang info
(define scribblings '(("manual.scrbl" ())))
```

- Run `raco setup` 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 collection that is installed as user-specific (e.g., the user package scope), the generated documentation is "doc/manual/index.html" within the collection directory. If the collection is installation-wide, however, then the documentation is generated as "manual/index.html" in the installation's "doc" directory.

4.1.2 Racket Typesetting and Hyperlinks

In the document source at the start of this chapter (§4.1.1 “Setting Up Library Documentation”), the Racket 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, add a `require` form like this:

```
@(require (for-label racket))
```

This `require` with `for-label` declaration introduces a document-time binding for each export of the `racket` module. When the document is built, the `racket` 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 `racket` 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/manual
@(require (for-label racket))
```

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

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

The `racket` form typesets a Racket expression for inline text, so it ignores the source formatting of the expression. The `racketblock` form, in contrast, typesets inset Racket code, and it preserves the expression's formatting from the document source.

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

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

Some example Racket code:

```
@racketblock[
(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")
]
```

4.1.3 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/manual
@(require (for-label racket))

@title{My Library}

Welcome to my documentation: @racket[(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.rkt" 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 `raco setup`. 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 Racket reference manual:

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

@title{My Library}

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

As mentioned in §4.1.2 “Racket Typesetting and Hyperlinks”, however, cross-document references based on `(require (for-label ...))` and `racket` are usually better than cross-document references using `secref`.

4.1.4 Defining Racket Bindings

Use `defproc` to document a procedure, `defform` 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 racket-based hyperlinks.

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

- Use `(require (for-label "helper.rkt"))` to import the binding information about the bindings of `"helper.rkt"` for use when typesetting identifiers. A relative reference `"helper.rkt"` 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.rkt"`, since the module path given to `defmodule` appears verbatim in the generated documentation.
- Use `defproc` to document the procedure.

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

```
#lang scribble/manual
@(require (for-label racket
            "helper.rkt"))
```

```

@title{My Library}

@defmodule[my-lib/helper]

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

  Replaces each @racket['cow] in @racket[lst] with
  @racket['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 `racket` 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.rkt"`, then the reference is hyperlinked to the definition above.

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

4.1.5 Showing Racket Examples

The examples form from `scribble/eval` helps you generate examples in your documentation.

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.rkt"` must be imported both via `(require (for-label ...))` and `require`:

```

#lang scribble/manual
@require scribble/eval ; <--- added
      "helper.rkt" ; <--- added
      (for-label racket
        "helper.rkt"))

@title{My Library}

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

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

Replaces each @racket['cow] in @racket[lst] with
@racket['aardvark].

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

```

4.1.6 Multi-Page Sections

Setting the `'multi-page` option (see §4.1.3 “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/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.rkt" 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 local table of contents is short, putting the introductory text after the call of `local-table-of-contents` may be appropriate.

4.2 Style Guide

Consistent style—for terms, typesetting, and prose—makes documentation clearer. As much as possible, follow the rules listed in this section. Many of the rules are arbitrary in the sense that a different choice of rule could work fine, but the only way to make our documentation consistent is to pick one of the choices.

There are too many rules to absorb easily on a first reading. Re-read this section after writing documentation for a library or two, and revisit the section periodically to refresh your memory and check for new rules.

4.2.1 Prose and Terminology

In the descriptive body of `def form`, `def proc`, etc., do not start with “This ...” Instead, start with a sentence whose implicit subject is the form or value being described (but only start the first sentence that way). 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 `racketmodname`). 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.”

Use the word “list” only when you mean a run-time value consisting of the empty list and `cons` cells; use the word “sequence” in other cases, if you must use any word. For example, do not write that `begin` has a “list of sub-forms;” instead, it has a “sequence of sub-forms.” Similarly, do not refer to a “list of arguments” in a function call; just write “arguments” if possible, or write “sequence of argument expressions.” (Unfortunately, “sequence” has acquired a specific run-time meaning, too, but the collision is less severe than the historical

confusion between lists and other entities in Lisp.)

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.

Hyphenate the words “sub-form” and “sub-expression.”

Use “Windows,” “Mac OS X,” and “Unix” for the three “platforms” (as opposed to “systems”) on which Racket runs. Use “Unix” as a generic term for Unix-like operating systems—notably including Linux—other than Mac OS X. Use “Unix” even when “Gtk” or “the X11 windowing system” would be more precisely correct, but use “X11” as adjective when necessary, such as “X11 display.” Racket runs “on” a platform, as opposed to “under” a platform.

4.2.2 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—always followed by `...+` in a grammar description. 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 `deftogether` 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*`.

For function arguments, use `v` as the meta-variable for “any value.” Use `x` as a meta-variable only for numerical values. Other conventions include `lst` for a list and `proc` for a procedure.

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

```
@racket[(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

```
@racket[(_rator-expr _rand-expr ...)]
```

which produces `(rator-expr rand-expr ...)`, where `rator-expr` and `rand-expr` are

typeset as meta-variables. The `defproc`, `defform`, etc. forms greatly reduce this burden in descriptions, since they automatically set up meta-variable typesetting for non-literal identifiers. In `defform`, 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 `racketidfont` (e.g., as in `rand-expr` above). Otherwise, use `litchar`, not merely `racketfont` or `verbatim`, to refer to a specific sequence of characters.

When a syntactic form synthesizes an identifier from a given identifier, use a combination of `racketidfont` and `racket` to describe the identifiers. For example, if `id` is combined with `is-` and `?` to form `is-id?`, then implement that identifier as `@racketidfont{is-}@racket[id]@racketidfont{?}`.

When using `defform` to describe a syntactic form, don't confuse the `#:contracts` clause with a grammar specification. Use `#:contracts` only for expressions within the syntactic form, and the contract is a run-time constraint—not a syntactic constraint, such as requiring a sub-form to be an identifier. Use `defform/subs` for syntactic constraints.

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

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

See also the `scribble/eval` library and §4.2.6 “Examples”.

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

4.2.3 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. 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 Racket variable name, since it is normally lowercase. For example, use “The *thing* argument is...” instead of “*thing* is...”

Use `etc` for “etc.” when it does not end a sentence, and include a comma after “etc.” unless it ends a sentence or is followed by other punctuation (such as a parenthesis).

4.2.4 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.

4.2.5 Indexing

Document and section titles, identifiers that are documented with `defproc`, `defform`, etc. are automatically indexed, as are terms defined with `deftech`.

Symbols are not indexed automatically. Use `indexed-racket` instead of `racket` for the instance of a symbol that roughly defines the use. For an example, try searching for “truncate” to find `'truncate` as used with `open-output-file`. Do not use something like `(index "'truncate")` to index a symbol, because it will not typeset correctly (i.e., in a fixed-width font with the color of a literal).

Use `index`, `as-index`, and `section-index` as a last resort. Create index entries for terms that are completely different from terms otherwise indexed. Do not try to index minor variations of a term or phrase in an attempt to improve search results; if search fails to find a word or phrase due to a minor variation, then the search algorithm should be fixed, not the index entry.

4.2.6 Examples

Strive to include examples (using `examples`) with the documentation of every function and syntactic form. When writing examples, refrain from using nonsense words like “foo” and “bar.” For example, when documenting `member`, resist the temptation to write

```
> (member "foo" '("bar" "foo" "baz"))
'("foo" "baz")
```

and instead write something like

```
> (member "Groucho" '("Harpo" "Groucho" "Zeppo"))
'("Groucho" "Zeppo")
```

4.3 Manual Forms

```
#lang scribble/manual      package: scribble-lib
```

The `scribble/manual` language provides all of `scribble/base` plus many additional functions that are specific to writing Racket documentation. It also associates style properties with the generated `doc` export to select the default Racket manual style for rendering; see §4.3.14 “Manual Rendering Style” for more information.

The `scribble/manual` name can also be used as a library with `require`, in which case it provides all of the same bindings, but without setting the reader or setting the default rendering format to the Racket manual format.

4.3.1 Typesetting Code

The `codeblock` and `code` forms (see §4.3.1.1 “#lang-Specified Code”) typeset code verbatim, adding a layer of color to the code based on the same syntax-coloring parsers that are used by DrRacket. Input that is parsed as an identifier is further given a lexical context and hyperlinked via `for-label` imports.

The `racketblock` and `racket` forms (see §4.3.1.2 “Racket Code”) typeset S-expression code roughly verbatim, but roughly by quoting the source term with `syntax`. Identifiers in the quoted S-expression are hyperlinked via `for-label` imports.

The two different approaches to typesetting code—`codeblock` and `code` versus `racketblock` and `racket`—have different advantages and disadvantages:

- The `codeblock` and `code` forms work with non-S-expression syntax, and they give authors more control over output (e.g., the literal number `2/4` is not normalized to `1/2`). The `codeblock` and `code` forms do not yet support escapes to Scribble element mode, and they cannot adapt spacing based on the width of elements in escapes.
- The `racketblock` and `racket` forms are more efficient and allow escapes to Scribble element mode. The `racketblock` and `racket` forms are tied to S-expression syntax, however, and they are based on a syntax representation that tends to normalize source terms (e.g., the literal number `2/4` is normalized to `1/2`).

#lang-Specified Code

```
(codeblock option ... str-expr ...+)
```

```

option = #:keep-lang-line? keep-expr
        | #:indent indent-expr
        | #:expand expand-expr
        | #:context context-expr
        | #:line-numbers line-number-expr
        | #:line-number-sep line-number-sep-expr

keep-expr : any/c
indent-expr : exact-nonnegative-integer?
expand-expr : (or/c #f (syntax? . -> . syntax?))
context-expr : syntax?
line-number-expr : (or/c #f exact-nonnegative-integer?)
line-number-sep-expr : exact-nonnegative-integer?

```

Parses the code formed by the strings produced by the *str-exprs* as a Racket module (roughly) and produces a block that typesets the code inset via `nested` with the style `'code-inset`.

The *str-exprs* should normally start with `#lang` to determine the reader syntax for the module, but the resulting “module” need not expand or compile—except as needed by *expand-expr*. If *expand-expr* is omitted or produces false, then the input formed by *str-expr* is read until an end-of-file is encountered, otherwise a single form is read from the input.

When *keep-expr* produces a true value (the default), the first line in the input (which is typically `#lang`) is preserved in the typeset output, otherwise the first line is dropped. The typeset code is indented by the amount specified by *indent-expr*, which defaults to 0.

When *expand-expr* produces `#f` (which is the default), identifiers in the typeset code are colored and linked based on for-label bindings in the lexical environment of the syntax object provided by *context-expr*. The default *context-expr* has the same lexical context as the first *str-expr*. When *line-number-expr* is true, line number is enabled starting from *line-number-expr*, and *line-number-sep* controls the separation (in spaces; defaults to 1) between the line numbers and code.

When *expand-expr* produces a procedure, it is used to macro-expand the parsed program, and syntax coloring is based on the parsed program.

For example,

```

@codeblock|{
  #lang scribble/manual
  @codeblock{
    #lang scribble/manual
    @title{Hello}
  }
}

```

```
}  
}|
```

produces the typeset result

```
#lang scribble/manual  
@codeblock{  
  #lang scribble/manual  
  @title{Hello}  
}
```

```
(codeblock0 option ... str-expr ...)
```

Like codeblock, but without the 'code-inset nested wrapper.

```
(code option ... str-expr ...)  
  
option = #:lang lang-line-expr  
        | #:expand expand-expr  
        | #:context context-expr  
  
lang-line-expr : (or/c #f string?)  
expand-expr   : (or/c #f (syntax? . -> . syntax?))  
context-expr  : syntax?
```

Like codeblock, but produces content instead of a block. No #lang line should appear in the string content; instead, it should be provided #:lang (as a string without "#lang ") if needed, and the #lang line is always stripped from the output when provided. Also, each newline in *str-exprs* is collapsed along with all surrounding whitespace to a single space.

For example,

```
This is @code[#:lang "at-exp racket"]|{@bold{Hi}}|'s result:  
@bold{Hi}.
```

produces the typeset result

```
This is @bold{Hi}'s result: Hi.
```

Racket Code

```
(racketblock maybe-escape datum ...)  
  
maybe-escape =  
  | #:escape escape-id
```

Typesets the *datum* sequence as a table of Racket code inset via `nested` with the style `'code-inset`. The source locations of the *datums* determine the generated layout. For example,

```
(racketblock
  (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 `racketblock`. Source-location span information is used to preserve `#true` versus `#t` and `#false` versus `#f`, and syntax-object properties are used to preserve square brackets and curly braces versus parentheses; otherwise, using syntax objects tends to normalize the form of S-expression elements, such as rendering `2/4` as `1/2`. When source-location information is not available, such as when it is lost by bytecode-compiled macros, spacing is inserted in the same style (within a single line) as the `racket` form.

In the above example, `define` is typeset as a keyword (in 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.

See also `quote-syntax/keep-srcloc` for use in a macro to preserve source-location information in a template.

Like other forms defined via `define-code`, `racketblock` expands identifiers that are bound as element transformers.

An `#:escape` clause specifies an identifier to escape back to an expression that produces an `element`. By default, the escape identifier is `unsyntax`. For example,

```
(racketblock
  (+ 1 #,(elem (racket x) (subscript "2"))))
```

produces

```
(+ 1 x2)
```

The `escape-id` that defaults to `unsyntax` is recognized via `free-identifier=?`, so a binding can hide the escape behavior:

```
(racketblock
  (let ([unsyntax #f])
    (racketblock
      #'(+ 1 #,x))))
```


The RACKETBLOCK form's default escape 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 Racket-typesetting mode using `unsyntax` and produces a paragraph using `t`:

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


(Note that `@#,foo{...}` reads as `#,(foo "...")`.)
- `code:blank` typesets as a blank space.
- `(code:hilite datum)` typesets like *datum*, but with a background highlight.
- `(code:quote datum)` typesets like `(quote datum)`, but without rendering the quote as `'`.
- `_id` typesets as *id*, but colored as a variable (like `racketvarfont`); 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.

See also `scribble/comment-reader`.

```
(RACKETBLOCK maybe-escape datum ...)
```

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

```
(racketblock0 maybe-escape datum ...)
```

Like `racketblock`, but without inseting the code via `nested`.

```
(RACKETBLOCK0 maybe-escape datum ...)
```

Like RACKETBLOCK, but without inseting the code via `nested`.

```
(racketresultblock maybe-escape datum ...)  
(racketresultblock0 maybe-escape datum ...)  
(RACKETRESULTBLOCK maybe-escape datum ...)  
(RACKETRESULTBLOCK0 maybe-escape datum ...)
```

Like `racketblock`, etc., but colors the typeset text as a result (i.e., a single color with no hyperlinks) instead of code.

```
(racketinput maybe-escape datum ...)  
(RACKETINPUT maybe-escape datum ...)
```

Like `racketblock` and `RACKETBLOCK`, but the *datums* are typeset after a prompt representing a REPL.

```
(racketinput0 maybe-escape datum ...)  
(RACKETINPUT0 maybe-escape datum ...)
```

Like `racketinput` and `RACKETINPUT`, but without inseting the code via `nested`.

```
(racketmod maybe-file maybe-escape lang datum ...)  
  
  maybe-file =  
    | #:file filename-expr  
  
  maybe-escape =  
    | #:escape escape-id
```

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

The source location of *lang* (relative to the body *datums*) determines the relative positioning of the `#lang` line in the typeset output. So, line up *lang* with the left end of the content code.

If `#:file` is provided, then the code block is typeset using `filebox` with *filename-expr* as the filename argument.

```
(racketmod0 maybe-file maybe-escape lang datum ...)
```

Like `racketmod`, but without inseting the code via `nested`.

```
(racket maybe-escape datum ...)
```

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

```
(RACKET maybe-escape datum ...)
```

Like `racket`, but with the `UNSYNTAX` escape like `racketblock`.

```
(racketresult maybe-escape datum ...)
```

Like racket, but typeset as a result (i.e., a single color with no hyperlinks).

```
(racketid maybe-escape datum ...)
```

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

```
(schemeblock maybe-escape datum ...)  
(SCHEMEBLOCK maybe-escape datum ...)  
(schemeblock0 maybe-escape datum ...)  
(SCHEMEBLOCK0 maybe-escape datum ...)  
(schemeinput maybe-escape datum ...)  
(schememod lang maybe-escape datum ...)  
(scheme maybe-escape datum ...)  
(SCHEME maybe-escape datum ...)  
(schemeresult maybe-escape datum ...)  
(schemeid maybe-escape datum ...)
```

Compatibility aliases. Each `scheme...` name is an alias for the corresponding `racket...` binding.

Preserving Comments

```
#reader scribble/comment-reader      package: scribble-lib
```

As a reader module, `scribble/comment-reader` reads a single S-expression that contains `;`-based comment lines, and it wraps the comments with `code:comment` for use with forms like `racketblock`. More precisely, `scribble/comment-reader` extends the current reader to adjust the parsing of `;`.

For example, within a Scribble document that imports `scribble/manual`,

```
@#reader scribble/comment-reader  
  (racketblock  
    ;; This is not a pipe  
    (make-pipe)  
  )
```

generates

```
; This is not a pipe  
(make-pipe)
```

The initial `@` is needed above to shift into S-expression mode, so that `#reader` is recognized as a reader declaration instead of literal text. Also, the example uses `(racketblock ...)` instead of `@racketblock[...]` because the `@`-reader would drop comments within the `racketblock` before giving `scribble/comment-reader` a chance to convert them.

The implementation of `scribble/comment-reader` uses `unsyntax` to typeset comments. When using `scribble/comment-reader` with, for instance, `RACKETBLOCK`, `unsyntax` does not escape, since `RACKETBLOCK` uses `UNSYNTAX` as its escape form. You can declare an escape identifier for `scribble/comment-reader` with `#:escape-id`. For example,

```
@#reader scribble/comment-reader #:escape-id UNSYNTAX
(RACKETBLOCK
  (define-syntax (m stx)
    (syntax-case stx ()
      [(_ x)
       ;; Well this was silly
       #`(#,x)]))
  )
```

generates

```
(define-syntax (m stx)
  (syntax-case stx ()
    [(_ x)
     ; Well this was silly
     #`(#,x)]))
```

Code Fonts and Styles

```
(racketmodname datum maybe-indirect)
(racketmodname (unsyntax expr) maybe-indirect)

maybe-indirect =
  | #:indirect
```

Like `racket`, but typeset as a module path. If *datum* is an identifier or *expr* produces a symbol, then it is hyperlinked to the module path's definition as created by `defmodule`.

If `#:indirect` is specified, then the hyperlink is given the `'indirect-link` style property, which makes the hyperlink's resolution in HTML potentially delayed; see `'indirect-link` for `link-element`.

```
(racketmodlink datum pre-content-expr ...)
```

Like `racketmod`, but separating the module path to link from the content to be linked. The *datum* module path is always linked, even if it is not an identifier.

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

Typesets *strs* 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.

```
(racketfont pre-content ...) → element?
  pre-content : pre-content?
```

The same as (`tt pre-content ...`), which applies the `'tt` style to immediate strings and symbols among the `pre-content` arguments. Beware that `pre-content` is `decoded` as usual, making `racketfont` a poor choice for typesetting literal code.

```
(racketplainfont pre-content ...) → element?
  pre-content : pre-content?
```

Applies the `'tt` style to `pre-content`. Beware that `pre-content` is `decoded` as usual, making `racketplainfont` a poor choice for typesetting literal code directly but useful for implementing code-formatting functions.

Added in version 1.6 of package `scribble-lib`.

```
(racketvalfont pre-content ...) → element?
  pre-content : pre-content?
```

Like `racketplainfont`, but colored as a value.

```
(racketresultfont [#:decode? decode?]
                  pre-content ...) → element?
  decode? : boolean? = #t
  pre-content : pre-content?
```

Like `racketplainfont`, but colored as a REPL result. When `decode?` is `#f`, then unlike `racketplainfont`, `racketresultfont` avoids `decoding` its argument.

```
(racketidfont pre-content ...) → element?
  pre-content : pre-content?
```

Like `racketplainfont`, but colored as an identifier.

```
(racketvarfont pre-content ...) → element?
  pre-content : pre-content?
```

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

```
(racketkeywordfont pre-content ...) → element?
  pre-content : pre-content?
```

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

```
(racketparenfont pre-content ...) → element?  
pre-content : pre-content?
```

Like `racketplainfont`, but colored like parentheses.

```
(racketmetafont pre-content ...) → element?  
pre-content : pre-content?
```

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

```
(racketcommentfont pre-content ...) → element?  
pre-content : pre-content?
```

Like `racketplainfont`, but colored as a comment.

```
(racketerror pre-content ...) → element?  
pre-content : pre-content?
```

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

```
(racketmodfont pre-content ...) → element?  
pre-content : pre-content?
```

Like `racketplainfont`, but colored as module name.

```
(racketoutput pre-content ...) → element?  
pre-content : pre-content?
```

Like `racketplainfont`, but colored as output.

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

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 `defproc` arrange for racket 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.

```
(schememodname datum)
(schememodname (unsyntax expr))
(schememodlink datum pre-content-expr ...)
(schemefont pre-content ...) → element?
  pre-content : pre-content?
(schemevalfont pre-content ...) → element?
  pre-content : pre-content?
(schemeresultfont pre-content ...) → element?
  pre-content : pre-content?
(schemeidfont pre-content ...) → element?
  pre-content : pre-content?
(schemevarfont pre-content ...) → element?
  pre-content : pre-content?
(schemekeywordfont pre-content ...) → element?
  pre-content : pre-content?
(schemeparenfont pre-content ...) → element?
  pre-content : pre-content?
(schememetafont pre-content ...) → element?
  pre-content : pre-content?
(schemeerror pre-content ...) → element?
  pre-content : pre-content?
(schememodfont pre-content ...) → element?
  pre-content : pre-content?
(schemeoutput pre-content ...) → element?
  pre-content : pre-content?
```

Compatibility aliases. Each `scheme...` name is an alias for the corresponding `racket...` binding.

4.3.2 Documenting Modules

```
(defmodule maybe-req one-or-multi option ... pre-flow ...)
```

```

maybe-req =
  | #:require-form content-expr

one-or-multi = module-spec
  | #:multi (module-spec ...+)

module-spec = module-path
  | content-expr

  option = #:module-paths (module-path ...)
  | #:no-declare
  | #:use-sources (src-module-path ...)
  | #:link-target? link-target?-expr
  | #:indirect
  | #:lang
  | #:reader
  | #:packages (pkg-expr ...)

```

Produces a sequence of flow elements (in a `splice`) to start the documentation for a module—or for multiple modules, if the `#:multi` form is used.

Each documented module specified as either a `module-path` (in the sense of `require`), in which case the module path is typeset using `racketmodname`, or by a `content-expr`. The latter case is triggered by the presence of a `#:module-paths` clause, which provides a plain `module-path` for each `module-spec`, and the plain `module-path` is used for cross-referencing.

If a `#:require-form` clause is provided and if `#:lang` and `#:reader` are not provided, the given expression produces content to use instead of `require` for the declaration of the module. The `#:require-form` clause is useful to suggest a different way of accessing the module instead of through `require`.

Besides generating text, unless `#:no-declare` appears as an option, this form expands to a use of `declare-exporting` with `module-paths`; the `#:use-sources` clause, if provided, is propagated to `declare-exporting`. Consequently, `defmodule` should be used at most once in a section without `#:no-declare`, though it can be shadowed with `defmodules` in sub-sections. Use `#:no-declare` 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

When `#:link-target?` is omitted or specified with an expression that produces a true value, then the `module-paths` are also declared as link targets though a `part-tag-decl` (which means that the `defmodule` form must appear before any sub-parts). These link targets are referenced via `racketmodname`, which thus points to the enclosing section, rather than the individual `module-paths`.

Specifying `#:indirect` normally makes sense only when `#:link-target?` is specified with a `#f` value. Specifying `#:indirect` makes the module path that is displayed (and that normally refers to some other declaration of the module) use `racketmodname` with `#:indirect`.

If `#:lang` is provided as an option, then the module name is shown after `#lang` (instead of in a `require` form) to indicate that the `module-paths` are suitable for use by either `require` or `#lang`. If the module path for `require` is syntactically different from the `#lang` form, use `#:module-paths` to provide the `require` variant (and make each `module-spec` a `content-expr`).

If `#:reader` is provided, then the module name is shown after `#reader` to indicate that the module path is intended for use as a reader module.

By default, the package (if any) that supplies the documented module is determined automatically, but a set of providing packages can be specified explicitly with `#:packages`. Each `pkg-expr` result is passed on to a function like `tt` for typesetting. Provide an empty sequence after `#:packages` to suppress any package name in the output. Each `pkg-expr` expression is duplicated for a `declare-exporting` form, unless `#:no-declare` is specified.

Each `option` form can appear at most once, and `#:lang` and `#:reader` are mutually exclusive.

The decoded `pre-flows` introduce the module, but need not include all of the module content.

```
(declare-exporting module-path ... maybe-pkgs maybe-sources)  
  
  maybe-pkgs =  
    | #:packages (pkg-expr ...)  
  
  maybe-sources =  
    | #:use-sources (module-path ...)
```

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

More significantly, the first `module-path` before `#:use-sources` plus the `module-paths` after `#:use-sources` determine the binding that is documented by each `deform`, `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 `module-path`.

- If `#:use-sources` *module-paths* are supplied, then they are tried in order before the first *module-path*. The *module-path* that provides 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 *module-path* in the `declare-exporting` declaration.

Use `#:use-sources` sparingly, but it is needed when

- bindings are documented as originating from a module *M*, but the bindings are actually re-exported from some module *P*; and
- other documented modules also re-export the bindings from *P*, but they are documented as re-exporting from *M*.

For example, the `parameterize` binding of `mzscheme` is documented as re-exported from `racket/base`, but `parameterize` happens to be implemented in a private module and re-exported by both `racket/base` and `mzscheme`. Importing `parameterize` from `mzscheme` does not go through `racket/base`, so a search for documentation on `parameterize` in `mzscheme` would not automatically connect to the documentation of `racket/base`. To make the connection, the documentation of `racket/base` declares the private module to be a source through `#:use-sources`, so that any re-export of `parameterize` from the private module connects to the documentation for `racket/base` (unless a re-export has its own documentation, which would override the automatic connection when searching for documentation).

The initial *module-paths* sequence can be empty if *module-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 *module-paths* in `#:use-sources` provide a binding context for connecting (via hyperlinks) definitions and uses of identifiers.

Supply `#:packages` to specify the package that provides the declared modules, which is otherwise inferred automatically from the first *module-path*. The package names are used, for example, by `history`.

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.

Changed in version 1.1 of package `scribble-lib`: Added `#:packages` clause.

```
(defmodulelang one-or-multi maybe-sources option ... pre-flow ...)
(defmodulelang one-or-multi #:module-path module-path
  option ... pre-flow ...)
```

Equivalent to `defmodule` with `#:lang`. The `#:module-path` *module-path* is provided, it is converted to `#:module-paths` (*module-path*).

```
(defmodulereader one-or-multi option ... pre-flow ...)
```

Equivalent to `defmodule` with `#:reader`.

```
(defmodule* maybe-req (module-spec ...+) option ... pre-flow ...)  
(defmodulelang* (module-spec ...+) option ... pre-flow ...)  
(defmodulereader* (module-spec ...+) option ... pre-flow ...)
```

Equivalent to `defmodule` variants with `#:multi`.

```
(defmodule*/no-declare maybe-req (module-spec ...) option ... pre-  
flow ...)  
(defmodulelang*/no-declare (module-spec ...) option ... pre-flow ...)  
(defmodulereader*/no-declare (module-spec ...) option ... pre-  
flow ...)
```

Equivalent to `defmodule` variants `#:no-declare`.

4.3.3 Documenting Forms, Functions, Structure Types, and Values

```
(defproc options prototype  
result-contract-expr-datum  
maybe-value  
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+

options = maybe-kind maybe-link maybe-id

maybe-kind =
            | #:kind kind-string-expr

maybe-link =
            | #:link-target? link-target?-expr

maybe-id =
            | #:id [src-id dest-id-expr]

maybe-value =
            | #:value value-expr-datum

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`. Unless *link-target?-expr* is specified and produces *#f*, the *id* is indexed, and it also registered so that `racket-typeset` uses of the identifier (with the same for-label binding) are hyper-linked to this documentation.

Examples:

```

@defproc[(make-sandwich [ingredients (listof ingredient?)])
  sandwich?]{
  Returns a sandwich given the right ingredients.
}

@defproc[#:kind "sandwich-maker"
  (make-reuben [ingredient sauerkraut?] ...
    [#:veggie? veggie? any/c #f])
  sandwich?]{

```

Produces a reuben given some number of `@racket[ingredient]`s.

If `@racket[veggie?]` is `@racket[#f]`, produces a standard reuben with corned beef. Otherwise, produces a vegetable reuben.

```
}
```

Renders like:

```
(make-sandwich ingredients) → sandwich?  
ingredients : (listof ingredient?)
```

Returns a sandwich given the right ingredients.

```
(make-reuben ingredient  
  ...  
  [#:veggie? veggie?]) → sandwich?  
ingredient : sauerkraut?  
veggie? : any/c = #f
```

Produces a reuben given some number of *ingredients*.

If *veggie?* is *#f*, produces a standard reuben with corned beef. Otherwise, produces a vegetable reuben.

When *id* is indexed and registered, 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 `racketblock0`.

```
(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 `racketblock0`, 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 `racket`, `racketblock`, 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. The information is formatted to fit (if possible) in the number of characters specified by the `current-display-width` parameter.

An optional `#:kind` specification chooses the decorative label, which defaults to "procedure". A `#f` result for `kind-string-expr` uses the default, otherwise `kind-string-expr` should produce a string. An alternate label should be all lowercase.

If `#:id [src-id dest-id-expr]` is supplied, then `src-id` is the identifier as it appears in the `prototype` (to be replaced by a defining instance), and `dest-id-expr` produces the identifier to be documented in place of `src-id`. This split between `src-id` and `dest-id-expr` roles is useful for functional abstraction of `defproc`.

If `#:value value-expr-datum` is given, `value-expr-datum` is typeset using `racketblock0` and included in the documentation. As a service to readers, please use `#:value` to document only simple, short functions.

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

Like `defproc`, but for multiple cases with the same `id`. 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 (but multiple `defprocs` grouped by `deftogether` also works for that case).

When an `id` has multiple calling cases, either they must be defined with a single `defproc*`, so that a single definition point exists for the `id`, or else all but one definition should use `#:link-target? #f`.

Examples:

```
@defproc*[[[(make-pb&j) sandwich?]  
           [(make-pb&j [jelly jelly?]) sandwich?]]]{  
  Returns a peanut butter and jelly sandwich. If @racket[jelly]  
  is provided, then it is used instead of the standard (grape)  
  jelly.  
}
```

Renders like:

```
(make-pb&j) → sandwich?  
(make-pb&j jelly) → sandwich?  
jelly : jelly?
```

Returns a peanut butter and jelly sandwich. If `jelly` is provided, then it is used instead of the standard (grape) jelly.

```
(defform options form-datum  
        maybe-grammar maybe-contracts  
        pre-flow ...)
```

```

    options = maybe-kind maybe-link maybe-id maybe-literals

maybe-kind =
  | #:kind kind-string-expr

maybe-link =
  | #:link-target? link-target?-expr

maybe-id =
  | #:id id
  | #:id [id id-expr]

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

maybe-grammar =
  | #:grammar ([nonterm-id clause-datum ...+] ...)

maybe-contracts =
  | #:contracts ([subform-datum contract-expr-datum]
                ...)

```

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

If `#:kind kind-string-expr` is supplied, it is used in the same way as for `defproc`, but the default kind is "syntax".

If `#:id [id id-expr]` is supplied, then *id* is the identifier as it appears in the *form-datum* (to be replaced by a defining instance), and *id-expr* produces the identifier to be documented. This split between *id* and *id-expr* roles is useful for functional abstraction of `defform`.

Unless *link-target?-expr* is specified and produces `#f`, the *id* (or result of *id-expr*) is indexed, and it is also registered so that `racket-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* (or result of *id-expr*), 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 `racket`, `racketblock`, 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).

If a `#:grammar` clause is provided, it includes an auxiliary grammar of non-terminals shown

with the *id* form. Each *nonterm-id* is specified as being any of the corresponding *clause-datums*.

If a `#:contracts` clause is provided, each *subform-datum* (typically an identifier that serves as a meta-variable in *form-datum* or *clause-datum*) is shown as producing a value that must satisfy the contract described by *contract-expr-datum*. Use `#:contracts` only to specify constraints on a *value* produced by an expression; for constraints on the *syntax* of a *subform-datum*, use grammar notation instead, possibly through an auxiliary grammar specified with `#:grammar`.

The typesetting of *form-datum*, *clause-datum*, *subform-datum*, and *contract-expr-datum* preserves the source layout, like racketblock.

Examples:

```
@defform[(sandwich-promise sandwich-expr)
         #:contracts ([sandwich-expr sandwich?)]]{
  Returns a promise to construct a sandwich. When forced, the promise
  will produce the result of @racket[sandwich-expr].
}

@defform[#:literals (sandwich mixins)
         (sandwich-promise* [sandwich sandwich-expr]
                           [mixins ingredient-expr ...])
         #:contracts ([sandwich-expr sandwich?]
                     [ingredient-expr ingredient?])]{
  Returns a promise to construct a sandwich. When forced, the promise
  will produce the result of @racket[sandwich-
  expr]. Each result of
  the @racket[ingredient-expr]s will be mixed into the resulting
  sandwich.
}

@defform[(sandwich-factory maybe-name factory-component ...)
         #:grammar
         [(maybe-name (code:line)
                       name)
          (factory-component (code:line #:protein protein-expr)
                            [vegetable vegetable-expr])]]{
  Constructs a sandwich factory. If @racket[maybe-
  name] is provided,
  the factory will be named. Each of the @racket[factory-
  component]
  clauses adds an additional ingredient to the sandwich pipeline.
}
```

Renders like:

```
(sandwich-promise sandwich-expr)

  sandwich-expr : sandwich?
```

Returns a promise to construct a sandwich. When forced, the promise will produce the result of *sandwich-expr*.

```
(sandwich-promise* [sandwich sandwich-expr]
                  [mixins ingredient-expr ...])

  sandwich-expr : sandwich?
  ingredient-expr : ingredient?
```

Returns a promise to construct a sandwich. When forced, the promise will produce the result of *sandwich-expr*. Each result of the *ingredient-exprs* will be mixed into the resulting sandwich.

```
(sandwich-factory maybe-name factory-component ...)

  maybe-name =
    | name

  factory-component = #:protein protein-expr
                    | [vegetable vegetable-expr]
```

Constructs a sandwich factory. If *maybe-name* is provided, the factory will be named. Each of the *factory-component* clauses adds an additional ingredient to the sandwich pipeline.

```
(deform* options [form-datum ...+]
  maybe-grammar maybe-contracts
  pre-flow ...)
```

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

Examples:

```
@deform*[((call-with-current-sandwich expr)
          (call-with-current-sandwich expr sandwich-handler-
            expr))]{
  Runs @racket[expr] and passes it the value of the current
  sandwich. If @racket[sandwich-handler-expr] is provided, its result
  is invoked when the current sandwich is eaten.
}
```

Renders like:

```
(call-with-current-sandwich expr)
(call-with-current-sandwich expr sandwich-handler-expr)
```

Runs *expr* and passes it the value of the current sandwich. If *sandwich-handler-expr* is provided, its result is invoked when the current sandwich is eaten.

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

Like `defform` with `#:link-target? #f`.

```
(defidform maybe-kind maybe-link id pre-flow ...)
```

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

```
(defidform/inline id)
(defidform/inline (unsyntax id-expr))
```

Like `defidform`, but *id* (or the result of *id-expr*, analogous to `defform`) is typeset as an inline element. Use this form sparingly, because the typeset form does not stand out to the reader as a specification of *id*.

```
(defsubform options form-datum
  maybe-grammar maybe-contracts
  pre-flow ...)
(defsubform* options [form-datum ...+]
  maybe-grammar maybe-contracts
  pre-flow ...)
```

Like `defform` and `defform*`, but with indenting on the left for both the specification and the *pre-flows*.

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

Like `defform` with `#:link-target? #f`, but with indenting on the left for both the specification and the *pre-flows*.

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

Similar to `deform` with `#:link-target? #f`, but without the initial identifier as an implicit literal, 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 `deform` 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.

```
(specspecsubform maybe-literals datum maybe-grammar maybe-contracts
 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.

```
(deform/subs options form-datum
 ([nonterm-id clause-datum ...+] ...)
 maybe-contracts
 pre-flow ...)
(deform*/subs options [form-datum ...+]
 ([nonterm-id clause-datum ...+] ...)
 maybe-contracts
 pre-flow ...)
(specform/subs maybe-literals datum
 ([nonterm-id clause-datum ...+] ...)
 maybe-contracts
 pre-flow ...)
(specsubform/subs maybe-literals datum
 ([nonterm-id clause-datum ...+] ...)
 maybe-contracts
 pre-flow ...)
(specspecsubform/subs maybe-literals datum
 ([nonterm-id clause-datum ...+] ...)
 maybe-contracts
 pre-flow ...)
```

Like `deform`, `deform*`, `specform`, `specsubform`, and `specspecsubform`, respectively, but the auxiliary grammar is mandatory and the `#:grammar` keyword is omitted.

Examples:

```
@deform/subs[(sandwich-factory maybe-name factory-component ...)
 [(maybe-name (code:line)
 name)
```

```

      (factory-component (code:line #:protein protein-
expr)
                        [vegetable vegetable-expr]))]{
  Constructs a sandwich factory. If @racket[maybe-
name] is provided,
  the factory will be named. Each of the @racket[factory-
component]
  clauses adds an additional ingredient to the sandwich pipeline.
}

```

Renders like:

```

(sandwich-factory maybe-name factory-component ...)

  maybe-name =
              | name
factory-component = #:protein protein-expr
                  | [vegetable vegetable-expr]

```

Constructs a sandwich factory. If *maybe-name* is provided, the factory will be named. Each of the *factory-component* clauses adds an additional ingredient to the sandwich pipeline.

```

(defparam maybe-link id arg-id
  contract-expr-datum
  maybe-value
  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.

Examples:

```

@defparam[current-sandwich sandwich sandwich?
  #:value empty-sandwich]{
  A parameter that defines the current sandwich for operations that
  involve eating a sandwich. Default value is the empty sandwich.
}

```

Renders like:

```

(current-sandwich) → sandwich?

```

```
(current-sandwich sandwich) → void?
  sandwich : sandwich?
= empty-sandwich
```

A parameter that defines the current sandwich for operations that involve eating a sandwich. Default value is the empty sandwich.

```
(defparam* maybe-link id arg-id
  in-contract-expr-datum out-contract-expr-datum
  maybe-value
  pre-flow ...)
```

Like `defparam`, but with separate contracts for when the parameter is being set versus when it is being retrieved (for the case that a parameter guard coerces values matching a more flexible contract to a more restrictive one; `current-directory` is an example).

```
(defboolparam maybe-link id arg-id
  maybe-value
  pre-flow ...)
```

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

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

  options = maybe-kind maybe-link maybe-id

  maybe-kind =
    | #:kind kind-string-expr

  maybe-link =
    | #:link-target? link-target?-expr

  maybe-id =
    | #:id id-expr

  maybe-value =
    | #:value value-expr-datum
```

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

If `#:kind kind-string-expr` is supplied, it is used in the same way as for `defproc`, but the default kind is `"value"`.

If `#:id id-expr` is supplied, then the result of `id-expr` is used in place of `id`.

If `#:value value-expr-datum` is given, `value-expr-datum` is typeset using `racket-block0` and included in the documentation. Wide values are put on a separate line.

Examples:

```
@defthing[moldy-sandwich sandwich?]{
  Don't eat this. Provided for backwards compatibility.
}

@defthing[empty-sandwich sandwich? #:value (make-sandwich empty)]{
  The empty sandwich.
}
```

Renders like:

```
| moldy-sandwich : sandwich?
```

```
|     Don't eat this. Provided for backwards compatibility.
```

```
| empty-sandwich : sandwich? = (make-sandwich empty)
```

```
|     The empty sandwich.
```

```
| (defstruct* maybe-link struct-name ([field-name contract-expr-datum] ...)
|   maybe-mutable maybe-non-opaque maybe-constructor
|   pre-flow ...)
```

```
| (defstruct maybe-link struct-name ([field-name contract-expr-datum] ...)
|   maybe-mutable maybe-non-opaque maybe-constructor
|   pre-flow ...)
```

```
|     maybe-link =
|       | #:link-target? link-target?-expr
```

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

```
|     maybe-mutable =
|       | #:mutable
```

```
|     maybe-non-opaque =
|       | #:prefab
|       | #:transparent
|       | #:inspector #f
```

```
|     maybe-constructor =
|       | #:constructor-name constructor-id
|       | #:extra-constructor-name constructor-id
|       | #:omit-constructor
```

Similar to `defform` or `defproc`, but for a structure definition. The `defstruct*` form corresponds to `struct`, while `defstruct` corresponds to `define-struct`.

Examples:

```
@defstruct[sandwich ([protein ingredient?] [sauce ingredient?])]{
  A structure type for sandwiches. Sandwiches are a pan-
  human foodstuff
  composed of a partially-enclosing bread material and various
  ingredients.
}
```

Renders like:

```
(struct sandwich (protein sauce)
  #:extra-constructor-name make-sandwich)
protein : ingredient?
sauce : ingredient?
```

A structure type for sandwiches. Sandwiches are a pan-human foodstuff composed of a partially-enclosing bread material and various ingredients.

```
(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.

Examples:

```
@deftogether[(@defthing[test-sandwich-1 sandwich?]
  @defthing[test-sandwich-2 sandwich?])]{
  Two high-quality sandwiches. These are provided for convenience
  in writing test cases
}
```

Renders like:

```
test-sandwich-1 : sandwich?
test-sandwich-2 : sandwich?
```

Two high-quality sandwiches. These are provided for convenience in writing test cases


```
(racketgrammar 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 racket.

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

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

```
(defidentifier id
  [#:form? form?
   #:index? index?
   #:show-libs? show-libs?]) → element?
id : identifier?
form? : boolean? = #f
index? : boolean? = #t
show-libs? : boolean? = #t
```

Typesets *id* as a Racket identifier, and also establishes the identifier as the definition of a binding in the same way as defproc, deform, etc. As always, the library that provides the identifier must be declared via defmodule or declare-exporting for an enclosing section.

If *form?* is a true value, then the identifier is documented as a syntactic form, so that uses of the identifier (normally including *id* itself) are typeset as a syntactic form.

If *index?* is a true value, then the identifier is registered in the index.

If *show-libs?* is a true value, then the identifier's defining module may be exposed in the typeset form (e.g., when viewing HTML and the mouse hovers over the identifier).

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

Compatibility aliases for racketgrammar and racketgrammar*.

```
(current-display-width) → exact-nonnegative-integer?
(current-display-width w) → void?
  w : exact-nonnegative-integer?
```

Specifies the target maximum width in characters for the output of defproc and defstruct.

4.3.4 Documenting Classes and Interfaces

```
(defclass maybe-link id super (intf-id ...) pre-flow ...)  
  
maybe-link =  
  | #:link-target? link-target?-expr  
  
  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 maybe-link 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 `'hidden` 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 maybe-mode maybe-link (id arg-spec ...)
  result-contract-expr-datum
  pre-flow ...)

maybe-link =
  | #:link-target? link-target?-expr

maybe-mode =
  | #:mode override
  | #:mode override-final
  | #:mode public-final
  | #:mode augment
  | #:mode augment-final
  | #:mode pubment
  | #:mode extend
  | #:mode extend-final
```

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

The `maybe-mode` specifies whether the method overrides a method from a superclass, and so on. (For these purposes, use `#:mode override` when refining a method of an implemented interface.) The `extend` mode is like `override`, but the description of the method should describe only extensions to the superclass implementation.

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

Like `defproc*`, but for a method within a `defclass` or `definterface` body. The *maybe-mode* specification is as in `defmethod`.

```
(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.

```
(this-obj)
```

Within a `defmethod` or similar form, typesets as a meta-variable that stands for the target of the method call. Use `(this-obj)` to be more precise than prose such as “this method’s object.”

4.3.5 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 `racket`.

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 : pre-flow?
```

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`.

4.3.6 Various String Forms

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

Like `elem`, but adds an `'aux` style property.

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

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 : pre-content?
```

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 : pre-content?
```

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

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

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

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

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

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

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

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

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

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

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

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

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

4.3.7 Links

See also §3.1.5 “Links”.

```
(racketlink id #:style style-expr pre-content ...)  
(racketlink id pre-content ...)  
  
id : identifier?  
pre-content : pre-content?
```

An element where the decoded *pre-content* is hyperlinked to the definition of *id*.

```
(schemelink id pre-content ...)
```

Compatibility alias for racketlink.

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

Alias of [hyperlink](#) for backward compatibility.

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

Alias of [other-doc](#) for backward compatibility.

```
(deftech pre-content
        ...
        [#:key key
         #:normalize? normalize?
         #:style? style?]) → element?
pre-content : pre-content?
key : (or/c string? #f) = #f
normalize? : any/c = #t
style? : any/c = #t
```

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

When *key* is *#f*, the *content->string* result of the decoded *pre-content* is used as a key for references. If *normalize?* is true, then the key string is normalized as follows:

- The string is case-folded.
- 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”.

If `style?` is true, then `defterm` is used on `pre-content`.

```
(tech pre-content
  ...
  [#:key key
   #:normalize? normalize?
   #:doc module-path
   #:tag-prefixes prefixes]) → element?
pre-content : pre-content?
key : (or/c string? #f) = #f
normalize? : any/c = #t
module-path : (or/c module-path? #f) = #f
prefixes : (or/c (listof string?) #f) = #f
```

Produces an element for the decoded `pre-content`, and hyperlinks it to the definition of the key as established by `deftech`. If `key` is false, the decoded content is converted to a string (using `content->string`) to use as a key; in either case, if `normalize?` is true, the key is normalized in the same way as for `deftech`. The `#:doc` and `#:tag-prefixes` arguments support cross-document and section-specific references, like in `secref`.

With the default style files, the hyperlink created by `tech` is somewhat quieter than most hyperlinks: the underline in HTML output is gray, instead of blue, and the term and underline turn blue 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
  ...
  [#:key key
   #:normalize? normalize?
   #:doc module-path
   #:tag-prefixes prefixes]) → element?
pre-content : pre-content?
key : (or/c string? #f) = #f
normalize? : any/c = #t
module-path : (or/c module-path? #f) = #f
prefixes : (or/c (listof string?) #f) = #f
```


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

4.3.8 Indexing

See also §3.1.6 “Indexing”.

```
(indexed-racket datum ...)
```

A combination of `racket` 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).

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

Compatibility alias for `indexed-racket`.

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

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 : pre-content?
```

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 : pre-content?
```

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 : pre-content?
```

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

4.3.9 Images

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

An alias for `image` for backward compatibility.

4.3.10 Bibliography

See also
`scribiblib/autobib`.

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

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

```
(bibliography [#:tag tag] entry ...) → part?
  tag : 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 : (or/c #f pre-content?)
  is-book? : boolean? = #f
  author : (or/c #f pre-content?) = #f
  location : (or/c #f pre-content?) = #f
  date : (or/c #f pre-content?) = #f
  url : (or/c #f pre-content?) = #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, or it is omitted if given as `#f`.
- *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, or it is omitted if given as `#f`.
- *date* is a date, usually just a year (as a string). It is typeset in the bibliography as given, or it is omitted if given as `#f`.
- *url* is an optional URL. It is typeset in the bibliography using `tt` and hyperlinked, or it is omitted if given as `#f`.

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

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

4.3.11 Version History

```
(history clause ...)
clause = #:added version-expr
        | #:changed version-expr content-expr
version-expr : valid-version?
content-expr : content?
```

Generates a block for version-history notes. The version refers to a package as determined by a `defmodule` or `declare-exporting` declaration within an enclosing section.

Normally, `history` should be used at the end of a `defform`, `defproc`, etc., entry, although it may also appear in a section that introduces a module (that has been added to a package). In the case of a `changed` entry, the content produced by `content-expr` should normally start with a capital letter and end with a period, but it can be a sentence fragment such as “Added a `#:changed` form.”

Added in version 1.1 of package `scribble-lib`.

4.3.12 Miscellaneous

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

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

`etc` : `element?`

Like "etc.", but with an abbreviation-ending period for use in the middle of a sentence.

`PLaneT` : `element?`

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

`manual-doc-style` : `style?`

A style to be used for a document's main `part` to get the style configuration of `#lang scribble/manual`. See §4.3.14 "Manual Rendering Style".

`(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>`.

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

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

`(inset-flow pre-flow ...)` → `nested-flow?`
`pre-flow` : `pre-flow?`

Creates a `nested-flow` with indenting on the left and right.

Using `nested` with the `'inset` style is a preferred alternative.

`(centerline pre-flow ...)` → `nested-flow?`
`pre-flow` : `pre-flow?`

An alias for `centered` for backward compatibility.

```
(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.
- When `_` appears before a non-empty sequence of numbers and letters, the sequence is typeset as a subscript.
- When `^` appears before a non-empty sequence of numbers and letters, the sequence is typeset as a superscript.

```
(filebox filename pre-flow ...) → block?  
filename : (or/c string? element?)  
pre-flow : pre-flow?
```

Typesets the *pre-flows* as the content of *filename*. For example, the content may be inset on the page with *filename* above it. If *filename* is a string, it is passed to `filepath` to obtain an `element`.

```
(deprecated [#:what what]  
  replacement  
  additional-notes ...) → block?  
what : content? = "library"  
replacement : content?  
additional-notes : content?
```

Produces an inset warning for deprecated libraries, functions, etc. (as described by *what*), where *replacement* describes a suitable replacement. The *additional-notes* are included after the initial deprecation message.

4.3.13 Index-Entry Descriptions

```
(require scribble/manual-struct) package: scribble-lib
```

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 ()
  #:extra-constructor-name make-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)
  #:extra-constructor-name make-exported-index-desc)
  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 ()
  #:extra-constructor-name make-form-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 ()
  #:extra-constructor-name make-procedure-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 ()
  #:extra-constructor-name make-thing-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 ()
  #:extra-constructor-name make-struct-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 ()
  #:extra-constructor-name make-class-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 ()
  #:extra-constructor-name make-interface-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 ()
  #:extra-constructor-name make-mixin-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)
  #:extra-constructor-name make-method-index-desc)
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.

```
(struct constructor-index-desc exported-index-desc (class-tag)
  #:extra-constructor-name make-constructor-index-desc)
class-tag : tag?
```

Indicates that the index entry corresponds to a constructor via `defconstructor` and `company`. The `name` field from `exported-index-desc` names the class or interface that contains the method. The `class-tag` field provides a pointer to the start of the documentation for the method's class or interface.

4.3.14 Manual Rendering Style

Using `#lang scribble/manual` for the main `part` of a document associates style properties on the `doc` export to select the Racket manual style for rendering.

A `html-defaults` style property is added to `doc`, unless `doc`'s style already has a `html-defaults` style property (e.g., supplied to `title`). Similarly, a `latex-default` style property is added if one is not already present. Finally, an `css-style-addition` property is always added.

For HTML rendering:

- The document's prefix file is set to "scribble-prefix.html", as usual, in [html-defaults](#).
- The document's style file is set to "manual-style.css" from the "scribble" collection in [html-defaults](#).
- The file "manual-files.css" from the "scribble" collection is designated as an additional accompanying file in [html-defaults](#).
- The file "manual-racket.css" from the "scribble" collection is added as a [css-style-addition](#).

To obtain this configuration without using `#lang scribble/manual`, use [manual-doc-style](#).

4.4 Racket

```
(require scribble/racket)    package: scribble-lib
(require scribble/scheme)
```

The `scribble/racket` library (or `scribble/scheme` for backward compatibility) provides utilities for typesetting Racket code. The `scribble/manual` forms provide a higher-level interface.

```
(define-code id typeset-expr)
(define-code id typeset-expr uncode-id)
(define-code id typeset-expr uncode-id d->s-expr)
(define-code id typeset-expr uncode-id d->s-expr stx-prop-expr)
```

Binds `id` to a form similar to `racket` or `racketblock` for typesetting code. The form generated by `define-code` handles source-location information, escapes via `unquote` by default, preserves binding and property information, and supports element transformers.

The supplied `typeset-expr` expression should produce a procedure that performs the actual typesetting. This expression is normally `to-element` or `to-paragraph`. The argument supplied to `typeset-expr` is normally a syntax object, but more generally it is the result of applying `d->s-expr`.

The optional `uncode-id` specifies the default escape from literal code to be recognized by `id`, and the default for `uncode-id` is `unsyntax`. A use of the `id` form can specify an alternate escape via `#:escape`, as in `racketblock` and `racket`.

The optional `d->s-expr` should produce a procedure that accepts three arguments suitable for `datum->syntax`: a syntax object or `#f`, an arbitrary value, and a vector for a source location. The result should record as much or as little of the argument information as needed by `typeset-expr` to typeset the code. Normally, `d->s-expr` is `datum->syntax`.

The `stx-prop-expr` should produce a procedure for recording a `'paren-shape` property when the source expression uses with `id` has such a property. The default is `syntax-property`.

```
(to-paragraph v
  [#:expr? expr?
   #:escapes? escapes?
   #:color? color?
   #:wrap-elem wrap-elem]) → block?
v : any/c
expr? : any/c = #f
escapes? : any/c = #t
color? : any/c = #t
wrap-elem : (element? . -> . element?) = (lambda (e) e)
```

Typesets an S-expression that is represented by a syntax object, where source-location information in the syntax object controls the generated layout. When source-location information is not available, default spacing is used (in the same single-line style as `to-element`).

Identifiers that have `for-label` bindings are typeset and hyperlinked based on definitions declared elsewhere (via `defproc`, `deform`, etc.). Unless `escapes?` is `#f`, the identifiers `code:line`, `code:comment`, `code:blank`, `code:hilite`, and `code:quote` are handled as in `racketblock`, as are identifiers that start with `_`.

In addition, the given `v` can contain `var-id`, `shaped-parens`, `just-context`, or `literal-syntax` structures to be typeset specially (see each structure type for details), or it can contain `element` structures that are used directly in the output.

If `expr?` is true, then `v` is rendered in expression style, much like `print` with the `print-as-expression` parameter set to `#t`. In that case, `for-label` bindings on identifiers are ignored, since the identifiers are all quoted in the output. Typically, `expr?` is set to true for printing result values.

If `color?` is `#f`, then the output is typeset without coloring.

The `wrap-elem` procedure is applied to each element constructed for the resulting block. When combined with `#f` for `color?`, for example, the `wrap-elem` procedure can be used to give a style to an element.

```
((to-paragraph/prefix prefix1
  prefix
  suffix)
 v
 [#:expr? expr?
  #:escapes? escapes?
  #:color? color?
  #:wrap-elem wrap-elem]) → block?
```

```

prefix1 : any/c
prefix  : any/c
suffix  : any/c
v       : any/c
expr?   : any/c = #f
escapes? : any/c = #t
color?  : any/c = #f
wrap-elem : (element? . -> . element?) = (lambda (e) e)

```

Like `to-paragraph`, but `prefix1` is prefixed onto the first line, `prefix` is prefix to any subsequent line, and `suffix` is added to the end. The `prefix1`, `prefix`, and `suffix` arguments are used as content, except that if `suffix` is a list of elements, it is added to the end on its own line.

```

(to-element v
  [#:expr? expr?
   #:escapes? escapes?
   #:defn? defn?]) → element?
v : any/c
expr? : any/c = #f
escapes? : any/c = #t
defn? : any/c = #f

```

Like `to-paragraph`, except that source-location information is mostly ignored, since the result is meant to be inlined into a paragraph. If `defn?` is true, then an identifier is styled as a definition site.

```

(to-element/no-color v
  [#:expr? expr?
   #:escapes? escapes?]) → element?
v : any/c
expr? : any/c = #f
escapes? : any/c = #t

```

Like `to-element`, but for-syntax bindings are ignored, and the generated text is uncolored. This variant is typically used to typeset results.

```

(struct var-id (sym)
  #:extra-constructor-name make-var-id)
sym : (or/c symbol? identifier?)

```

When `to-paragraph` and variants encounter a `var-id` structure, it is typeset as `sym` in the variable font, like `racketvarfont`—unless the `var-id` appears under quote or quasiquote, in which case `sym` is typeset as a symbol.

```
(struct shaped-parens (val shape)
  #:extra-constructor-name make-shaped-parens)
val : any/c
shape : char?
```

When `to-paragraph` and variants encounter a `shaped-parens` structure, it is typeset like a syntax object that has a `'paren-shape` property with value `shape`.

```
(struct long-boolean (val)
  #:extra-constructor-name make-long-boolean)
val : boolean?
```

When `to-paragraph` and variants encounter a `long-boolean` structure, it is typeset as `#true` or `#false`, as opposed to `#t` or `#f`.

```
(struct just-context (val context)
  #:extra-constructor-name make-just-context)
val : any/c
context : syntax?
```

When `to-paragraph` and variants encounter a `just-context` structure, it is typeset using the source-location information of `val` just the lexical context of `ctx`.

```
(struct literal-syntax (stx)
  #:extra-constructor-name make-literal-syntax)
stx : any/c
```

When `to-paragraph` and variants encounter a `literal-syntax` structure, it is typeset as the string form of `stx`. This can be used to typeset a syntax-object value in the way that the default printer would represent the value.

```
(element-id-transformer? v) → boolean?
v : any/c
```

Provided for-syntax; returns `#t` if `v` is an element transformer created by `make-element-id-transformer`, `#f` otherwise.

```
(make-element-id-transformer proc) → element-id-transformer?
proc : (syntax? . -> . syntax?)
```

Provided for-syntax; creates an *element transformer*. When an identifier has a transformer binding to an element transformer, then forms generated by `define-code` (including `racket` and `racketblock`) typeset the identifier by applying the `proc` to the identifier. The result must be an expression whose value, typically an `element`, is passed on to functions like `to-paragraph`.

```
(variable-id? v) → boolean?  
  v : any/c
```

Provided for-syntax; returns `#t` if `v` is an element transformer created by `make-variable-id`, `#f` otherwise.

```
(make-variable-id sym) → variable-id?  
  sym : (or/c symbol? identifier?)
```

Provided for-syntax; like `make-element-id-transformer` for a transformer that produces `sym` typeset as a variable (like `racketvarfont`)—unless it appears under quote or quasiquote, in which case `sym` is typeset as a symbol.

```
output-color : style?  
input-color : style?  
input-background-color : style?  
no-color : style?  
reader-color : style?  
result-color : style?  
keyword-color : style?  
comment-color : style?  
paren-color : style?  
meta-color : style?  
value-color : style?  
symbol-color : style?  
variable-color : style?  
opt-color : style?  
error-color : style?  
syntax-link-color : style?  
value-link-color : style?  
module-color : style?  
module-link-color : style?  
block-color : style?  
highlighted-color : style?
```

Styles that are used for coloring Racket programs, results, and I/O.

4.5 Evaluation and Examples

```
(require scribble/eval)    package: scribble-lib
```

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 maybe-eval maybe-escape datum ...)

  maybe-eval =
    | #:eval eval-expr
  maybe-escape =
    | #:escape escape-id
```

Like `racketinput`, 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-expr` is not provided, an evaluator is created using `make-base-eval`. See also `make-eval-factory`.

If the value of `current-print` in the sandbox is changed from its default value, or if `print-as-expression` in the sandbox is set to `#f`, then each evaluation result is formatted to a port by applying (`current-print`) to the value; the output port is set to a pipe that supports specials in the sense of `write-special`, and non-character values written to the port are used as content. Otherwise, when the default `current-print` is in place, result values are typeset using `to-element/no-color`.

Certain patterns in `datum` are treated specially:

- A `datum` of the form `(code:line code-datum (code:comment comment-datum ...))` is treated as `code-datum` for evaluation.
- Other uses of `code:comment` and `code:blank` are stripped from each `datum` before evaluation.
- A `datum` of the form `(eval:alts show-datum eval-datum)` is treated as `show-datum` for typesetting and `eval-datum` for evaluation.
- A `datum` of the form `(eval:check eval-datum expect-datum)` is treated like `eval-datum`, but `check-datum` is also evaluated, and an error is raised if they are not `equal?`.
- A `datum` of the form `(eval:result content-expr out-expr err-expr)` involves no sandboxed evaluation; instead, the content result of `content-expr` is used as the typeset form of the result, `out-expr` is treated as output printed by the expression, and `err-expr` is error output printed by the expression. The `out-expr` and/or `err-expr` can be omitted, in which case they default to empty strings.

Normally, `eval:result` is used in the second part of an `eval:alts` combination.

- A *datum* of the form `(eval:results content-list-expr out-expr err-expr)` is treated like an `eval:result` form, except that *content-list-expr* should produce a list of content for multiple results of evaluation. As with `eval:result`, *out-expr* and *err-expr* are optional.

As an example,

```
#lang scribble/manual
@(require racket/sandbox
          scribble/eval)
@(define my-evaluator
  (parameterize ([sandbox-output 'string]
                [sandbox-error-output 'string])
    (make-evaluator 'typed/racket/base)))
@interaction[#:eval my-evaluator

      (: my-sqr (Real -> Real))
      (define (my-sqr x)
        (* x x))
      (my-sqr 42)]
```

uses an evaluator whose language is `typed/racket/base`.

```
| (interaction0 maybe-eval maybe-escape datum ...)
```

Like `interaction`, but without inseting the code via `nested`.

```
| (interaction/no-prompt maybe-eval maybe-escape datum)
```

Like `interaction`, but does not render the output with a prompt.

```
| (interaction-eval maybe-eval maybe-escape datum)
```

Like `interaction-eval`, evaluates the quoted form of *datum*, but returns the empty string and does not catch errors.

```
| (interaction-eval-show maybe-eval maybe-escape datum)
```

Like `interaction-eval`, but produces an element representing the printed form of the evaluation result.

```
| (racketblock+eval maybe-eval maybe-escape datum ...)
```

Combines `racketblock` and `interaction-eval`.

```
(racketblock0+eval maybe-eval maybe-escape datum ...)
```

Combines racketblock0 and interaction-eval.

```
(racketmod+eval maybe-eval maybe-escape name datum ...)
```

Combines racketmod and interaction-eval.

```
(def+int maybe-eval maybe-escape defn-datum expr-datum ...)
```

Like interaction, except the *defn-datum* is typeset as for racketblock (i.e., no prompt) and a line of space is inserted before the *expr-datums*.

```
(defs+int maybe-eval maybe-escape (defn-datum ...) expr-datum ...)
```

Like def+int, but for multiple leading definitions.

```
(examples maybe-eval maybe-escape datum ...)
```

Like interaction, but with an “Examples:” label prefixed.

```
(defexamples maybe-eval maybe-escape 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 [#:pretty-print? pretty-print?
                #:lang lang]
  input-program ...)
→ (any/c . -> . any)
pretty-print? : any/c = #t
              (or/c module-path?
lang :        (list/c 'special symbol?) = '(begin)
              (cons/c 'begin list?))
input-program : any/c
```

Creates an evaluator using (`make-evaluator 'racket/base #:lang lang input-program ...`), setting sandbox parameters to disable limits, setting the outputs to `string`, and not adding extra security guards.

If `pretty-print?` is true, the sandbox’s printer is set to `pretty-print-handler`. In that case, values that are convertible in the sense of `convertible?` are printed using `write-special`, except that values that are serializable in the sense of `serializable?` are serialized for transfers from inside the sandbox to outside (which can avoid pulling code and support from the sandboxed environment into the document-rendering environment).

Changed in version 1.6 of package `scribble-lib`: Changed treatment of convertible values that are serializable.

```
(make-base-eval-factory mod-paths
                        [#:pretty-print? pretty-print?
                         #:lang lang])
→ (-> (any/c . -> . any))
mod-paths : (listof module-path?)
pretty-print? : any/c = #t
              (or/c module-path?
                 (list/c 'special symbol?) = '(begin)
                 (cons/c 'begin list?))
```

Produces a function that is like `make-base-eval`, except that each module in `mod-paths` is attached to the evaluator's namespace. The modules are loaded and instantiated once (when the returned `make-base-eval`-like function is called the first time) and then attached to each evaluator that is created.

```
(make-eval-factory mod-paths
                  [#:pretty-print? pretty-print?
                   #:lang lang])
→ (-> (any/c . -> . any))
mod-paths : (listof module-path?)
pretty-print? : any/c = #t
              (or/c module-path?
                 (list/c 'special symbol?) = '(begin)
                 (cons/c 'begin list?))
```

Like `make-base-eval-factory`, but each module in `mod-paths` is also required into the top-level environment for each generated evaluator.

```
(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.


```
(scribble-exn->string) → (-> (or/c exn? any/c) string?)
(scribble-exn->string handler) → void?
  handler : (-> (or/c exn? any/c) string?)
```

A parameter that controls how exceptions are rendered by interaction. Defaults to

```
(λ (e)
  (if (exn? e)
      (exn-message e)
      (format "uncaught exception: ~s" e)))
```

```
(with-eval-preserve-source-locations expr ...)
```

By default, the evaluation forms provided by this module, such as `interaction` and `examples`, discard the source locations from the expressions they evaluate. Within a `with-eval-preserve-source-locations` form, the source locations are preserved. This can be useful for documenting forms that depend on source locations, such as Redex's typesetting macros.

4.6 In-Source Documentation

The `scribble/srcdoc` and `scribble/extract` libraries support writing documentation within the documented code along with an export contract, similar to using JavaDoc. With this approach, a single contract specification is used both for the run-time contract and the documentation of an exported binding.

The `scribble/srcdoc` library provides forms for exporting a binding with associated documentation. The `scribble/extract` library is used to pull `scribble/srcdoc`-based documentation into a Scribble document (perhaps for multiple libraries).

Although documentation is written with a library's implementation when using `scribble/srcdoc`, the documentation creates no run-time overhead for the library. Similarly, typesetting the documentation does not require running the library. The two phases (run time versus documentation time) are kept separate in much the same way that the module system keeps expansion-time code separate from run-time code, and documentation information is recorded in a submodule to be separately loadable from the enclosing module.

For an example use, see the "file" collection's "gif.rkt" source file and the corresponding extraction in "scriblings/gif.scrbl". As that example illustrates, starting the module declaration with

```
#lang at-exp
```

enables the `@`-reader, which is handy for writing documentation expressions.

4.6.1 Source Annotations for Documentation

```
(require scribble/srcdoc)    package: scribble-lib
```

Documentation information generated by `scribble/srcdoc` forms are accumulated into a `srcdoc` submodule. The generated submodule is accessed by the bindings of `scribble/extract`.

```
(for-doc require-spec ...)
```

A `require` sub-form for bindings that are needed at documentation time (and documentation-expansion time, etc.) instead of run time (and expansion time, etc.). A `for-doc` import has no effect on a normal use of the library; it affects only documentation extraction.

Typically, a library that uses `scribble/srcdoc` includes at least `(require (for-doc scribble/base scribble/manual))` to get core Racket forms and basic Scribble functions to use in documentation expressions.

```
(proc-doc/names id contract arg-specs (desc-expr ...))

arg-specs = ((arg-id ...) ((arg-id default-expr) ...))
           | (arg-id ...)

contract = (-> arg ... result)
           | (->* (mandatory ...) (optional ...) result)
           | (case-> (-> arg ... result) ...)

mandatory = contract-expr
           | keyword contract-expr

optional = contract-expr
          | keyword contract-expr
```

A `provide` sub-form that exports `id` with the contract described by `contract` just like using `contract-out`.

The `arg-spec` specifies the names of arguments and the default values, which are not normally written as part of a contract. They are combined with the contract expression to generate the description of the binding in the documentation via `defproc`. The `(arg-id default-expr)` pairs specify the names and default values of the optional arguments. If the contract supports optional arguments, then the first `arg-specs` form must be used, otherwise the second must be used.

The `desc-expr` is a sequence of documentation-time expressions that produces prose to describe the exported binding—that is, the last part of the generated `defproc`, so the description can refer to the `arg-ids` using `racket`.

The normal requires of the enclosing library are effectively converted into for-label requires when generating documentation, so that identifiers in the `contracts` are linked to their corresponding documentation. Similarly, any binding that is available in the run-time phase of the enclosing library can be referenced in documentation prose using the racket form.

```
(proc-doc id contract maybe-defs (desc-expr ...))

  contract = (-> result)
             | (->i (arg ...) (opt ...) maybe-pre [id res])
             | (->i (arg ...) (opt ...) maybe-pre (values [id res] ...))
             | (->i (arg ...) (opt ...) #:rest rest [id result-expr])
             | (->d (arg ...) () maybe-pre (values [id result] ...))
             | (->d (arg ...) () maybe-pre [id result])
             | (->d (arg ...) () #:rest id rest [id result])

  maybe-pre =
             | #:pre (pre-id ...) condition

  maybe-defs =
             | (default-expr default-expr ...)
```

Like `proc-doc/names`, but supporting contract forms that embed argument identifiers. Only a subset of `->i` and `->d` forms are currently supported.

If the sequence of optional arguments, `(opt ...)` is empty then the `maybe-arg-desc` must not be present. If it is non-empty, then it must have as many default expressions as there are optional arguments.

```
(thing-doc id contract-expr (desc-expr ...))
```

Like `proc-doc`, but for an export of an arbitrary value.

```
(parameter-doc id (parameter/c contract-expr) arg-id (desc-expr ...))
```

Like `proc-doc`, but for exporting a parameter.

```
(struct*-doc struct-name
  ([field-name contract-expr-datum] ...)
  maybe-omit-constructor
  maybe-mutable maybe-non-opaque maybe-constructor
  (desc-expr ...))

  maybe-omit-constructor =
  | #:omit-constructor
```

Like `proc-doc`, but for struct declarations that use `struct`.

The `maybe-mutable`, `maybe-non-opaque`, and `maybe-constructor` options are as in `defstruct`.

```
(struct-doc struct-name
  ([field-name contract-expr-datum] ...)
  maybe-omit-constructor
  maybe-mutable maybe-non-opaque maybe-constructor
  (desc-expr ...))
```

Like `struct*-doc`, but for struct declarations that use `define-struct`.

```
(form-doc options form-datum
  maybe-grammar maybe-contracts
  (desc-expr ...))

  options = maybe-kind maybe-link maybe-id maybe-literals

  maybe-kind =
  | #:kind kind-string-expr

  maybe-link =
  | #:link-target? link-target?-expr

  maybe-id =
  | #:id id
  | #:id [id id-expr]

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

  maybe-grammar =
  | #:grammar ([nonterm-id clause-datum ...+] ...)

  maybe-contracts =
  | #:contracts ([subform-datum contract-expr-datum]
  ...)
```

Like `proc-doc`, but for an export of a syntactic form. If `#:id` is provided, then `id` is the exported identifier, otherwise the exported identifier is extracted from `form-datum`.

See `defform` for information on `options`, `form-datum`, `maybe-grammar`, and `maybe-contracts`.

Added in version 1.6 of package `scribble-lib`.

| `(begin-for-doc form ...)`

Like to `begin-for-syntax`, but for documentation time instead of expansion time. The *forms* can refer to binding required with `for-doc`.

For example, a definition in `begin-for-doc` can be referenced by a *desc-expr* in `proc-doc/names`.

| `(generate-delayed-documents)`

Causes documentation information to be recorded as a macro that is expanded (along with any `for-doc` imports) in the module that uses `include-extracted` or `provide-extracted`, instead of within (a submodule of) the module that declares the information.

Delaying document generation in this way allows `(for-doc (for-label ...))` imports that would otherwise create cyclic module dependencies.

To avoid problems with accumulated `for-doc` imports across modules, `generate-delayed-documents` declaration should appear before any `for-doc` import.

| `(require/doc require-spec ...)`

A legacy shorthand for `(require (for-doc require-spec ...))`.

| `(provide/doc spec ...)`

A legacy alternative to `(provide spec ...)`

4.6.2 Extracting Documentation from Source

```
(require scribble/extract)    package: scribble-lib
```

| `(include-extracted module-path)`

Expands to a sequence of documentation forms extracted from *module-path*, which is expected to be a module that uses `scribble/srcdoc` (so that the module has a `srcdoc` submodule).

| `(provide-extracted module-path)`

Similar to `include-extracted`, but the documentation is packaged and exported as `exported`, instead of left inline.

Use this form in combination with `include-previously-extracted` when documentation from a single source is to be split and typeset among multiple documentation locations. The `provide-extracted` form extracts the documentation once, and then `include-previously-extracted` form extracts documentation for specific bindings as needed.

```
(include-previously-extracted module-path regexp)
```

Similar to `include-extracted`, but instead of referring to the source that contains its own documentation, `module-path` refers to a module that uses `provide-extracted`. The `include-previously-extracted` form expands to documentation forms for all identifiers whose string forms match `regexp`.

4.7 BNF Grammars

```
(require scribble/bnf)      package: scribble-lib
```

The `scribble/bnf` library provides utilities for typesetting grammars.

For example,

```
@(let ([open @litchar{()}
        [close @litchar{)}])
      @BNF[(list @nonterm{expr}
                @nonterm{id}
                @BNF-seq[open @kleenepus[@nonterm{expr}] close]
                @BNF-seq[open @litchar{lambda}
                          open @kleenestar[@nonterm{id}] close
                          @nonterm{expr} close]
                @nonterm{val}]
          (list @nonterm{val}
                @BNF-alt[@nonterm{number} @nonterm{primop}])
          (list @nonterm{id}
                @elem{any name except for @litchar{lambda}})])
```

produces the output

```
⟨expr⟩ ::= ⟨id⟩
         | ( ⟨expr⟩+ )
         | ( lambda ( ⟨id⟩* ) ⟨expr⟩ )
         | ⟨val⟩
⟨val⟩   ::= ⟨number⟩ | ⟨primop⟩
⟨id⟩    ::= any name except for lambda
```

See also `racketgrammar`.

```
(BNF prod ...) → table?
```

```
| prod : (cons element? (listof (or/c block? 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-seq-lines elems ...) → block?  
| elems : (listof element?)
```

Typesets a sequence that is broken into multiple lines, where each `elems` is one line.

```
| (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.

```
| (kleenepplus 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.

4.8 Compatibility Libraries

4.8.1 Compatibility Structures And Processing

```
(require scribble/struct) package: scribble-lib
```

The `scribble/struct` compatibility library mostly re-exports `scribble/core`, but using some different names (e.g., `blockquote` instead of `nested-flow`).

The following structure types and functions are re-exported directly:

```
collect-info resolve-info tag? block?  
delayed-block collected-info delayed-element  
part-relative-element collect-info-parents  
collect-element render-element generated-tag  
tag-key content->string element->string  
block-width element-width  
info-key? part-collected-info collect-put!  
resolve-get resolve-get/tentative resolve-get/ext?  
resolve-search resolve-get-keys
```

The following structure types are re-exported, but the constructors and some selectors are replaced as documented further below:

```
part paragraph table itemization compound-paragraph  
element toc-element target-element toc-target-element toc-target2-  
element  
page-target-element redirect-target-element link-element  
index-element
```

Several additional compatibility functions and structure types are also exported.


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

For backward compatibility. Compared to the normal constructor for `part`, parses `style` to convert old formats to the current one. Also, if `title-content` is a list with a single item, the item by itself is stored in the resulting `part`.

```
(part-flow p) → (listof block?)
p : part?
```

For backward compatibility. An alias for `part-blocks`.

```
(part-title-content p) → list?
p : part?
```

For backward compatibility. Like the normal selector, but if the result would not be a list, it is coerced to one.

```
(make-versioned-part tag-prefix
                    tags
                    title-content
                    style
                    to-collect
                    blocks
                    parts
                    version) → part?
tag-prefix : (or/c false/c string?)
tags : (listof tag?)
title-content : (or/c false/c list?)
style : any/c
to-collect : list?
blocks : (listof block?)
parts : (listof part?)
version : string?
```

```
(versioned-part? v) → boolean?  
v : any/c
```

For backward compatibility. Like `make-part`, but adds a the `document-version` style property using the given `version`. The `versioned-part?` predicate recognizes a `part` with a `document-version` property.

```
(make-unnumbered-part tag-prefix  
                      tags  
                      title-content  
                      style  
                      to-collect  
                      blocks  
                      parts) → part?  
tag-prefix : (or/c false/c string?)  
tags : (listof tag?)  
title-content : (or/c false/c list?)  
style : any/c  
to-collect : list?  
blocks : (listof block?)  
parts : (listof part?)  
(unnumbered-part? v) → boolean?  
v : any/c
```

For backward compatibility. Like `make-part`, but adds the `'unnumbered` style property. The `unnumbered-part?` predicate recognizes a `part` with the `'unnumbered` property.

```
(make-paragraph content) → paragraph?  
content : list?
```

For backward compatibility. Compared to the normal constructor for `paragraph`, omits a style argument. Also, if `content` is a list containing a single item, the item by itself is stored in the resulting `paragraph`.

```
(paragraph-content p) → list?  
p : paragraph?
```

For backward compatibility. Like the normal selector, but if the result would not be a list, it is coerced to one.

```
(make-styled-paragraph content style) → paragraph?  
content : list?  
style : any/c  
(styled-paragraph? v) → boolean?  
v : any/c  
(styled-paragraph-style p) → style?  
p : paragraph?
```

For backward compatibility. Compared to the normal constructor for `paragraph`, parses `style` to convert old formats to the current one. The `styled-paragraph?` predicate and `styled-paragraph-style` accessor are aliases for `paragraph?` and `paragraph-style`.

```
(make-omitable-paragraph content) → paragraph?
  content : list?
(omitable-paragraph? v) → boolean?
  v : any/c
```

For backward compatibility. Like `make-paragraph`, but adds the `'omitable` style property. The `omitable-paragraph?` predicate checks for a paragraph with the property.

```
(make-table style blocksss) → table?
  style : any/c
  blocksss : (listof (listof (or/c (listof block?) (one-of/c 'cont))))
```

For backward compatibility. Compared to the normal constructor for `table`, the style is converted, and each cell has a list of blocks instead of a single block. If any such list has multiple blocks, they are combined into a `nested-flow`.

```
(table-flowss table)
→ (listof (listof (or/c (listof block?) (one-of/c 'cont))))
  table : table?
```

For backward compatibility. Like `table-blockss`, but adds a list wrapper to be consistent with `make-table`.

```
(make-itemization blockss) → itemization?
  blockss : (listof (listof block?))
```

For backward compatibility. Compared to the normal constructor for `itemization`, omits a style argument.

```
(make-styled-itemization style blockss) → itemization?
  style : any/c
  blockss : (listof (listof block?))
(styled-itemization? v) → boolean?
  v : any/c
(styled-itemization-style i) → style?
  i : itemization?
```

For backward compatibility. Compared to the normal constructor for `itemization`, parses `style` to convert old formats to the current one. The `styled-itemization?` predicate is an alias for `itemization?`, and `styled-itemization-style` is an alias for `itemization-style`.

```
(make-blockquote style blocks) → nested-flow?
  style : any/c
  blocks : (listof block?)
```

For backward compatibility. Like `make-nested-flow`, but `style` is parsed to the current format.

```
(make-auxiliary-table style blocksss) → table?
  style : any/c
  blocksss : (listof (listof (or/c (listof block?) (one-of/c 'cont))))
(auxiliary-table? v) → boolean?
  v : any/c
```

For backward compatibility. Like `make-table`, but adds the `'aux` style property. The `auxiliary-table?` predicate recognizes tables with the `'aux` property.

```
(make-compound-paragraph style blocks) → compound-paragraph?
  style : any/c
  blocks : (listof block?)
```

For backward compatibility. Compared to the normal constructor for `compound-paragraph`, parses `style` to convert old formats to the current one.

```
(make-element style content) → element?
  style : any/c
  content : list?
(make-toc-element style content toc-content) → toc-element?
  style : any/c
  content : list?
  toc-content : list?
(make-target-element style content tag) → target-element?
  style : any/c
  content : list?
  tag : tag?
(make-toc-target-element style content tag) → toc-target-element?
  style : any/c
  content : list?
  tag : tag?
(make-toc-target2-element style
                          content
                          tag
                          toc-content) → toc-target2-element?
  style : any/c
  content : list?
  tag : tag?
  toc-content : content?
```

```

(make-page-target-element style content tag)
→ page-target-element?
  style : any/c
  content : list?
  tag : tag?
(make-redirect-target-element style
                               content
                               tag
                               alt-path
                               alt-anchor)
→ redirect-target-element?
  style : any/c
  content : list?
  tag : tag?
  alt-path : path-string?
  alt-anchor : string?
(make-link-element style content tag) → link-element?
  style : any/c
  content : list?
  tag : tag?
(make-index-element style
                    content
                    tag
                    plain-seq
                    entry-seq
                    desc) → index-element?
  style : any/c
  content : list?
  tag : tag?
  plain-seq : (and/c pair? (listof string?))
  entry-seq : list?
  desc : any/c

```

For backward compatibility. Compared to the normal constructors, parses *style* to convert old formats to the current one.

```

(element? v) → boolean?
  v : any/c
(element-content e) → list?
  e : element?
(element-style e) → element-style?
  e : element?

```

For backward compatibility. A content list is treated as an element by these functions, and the result of `element-content` is always a list.

```
(make-aux-element style content) → element?
  style : any/c
  content : list?
```

For backward compatibility. Like `make-element`, but adds the `'aux` style property.

```
(make-hover-element style content text) → element?
  style : any/c
  content : list?
  text : string?
```

For backward compatibility. Like `make-element`, but adds `hover-property` containing `text` to the element's style.

```
(make-script-element style
                      content
                      type
                      script) → element?
  style : any/c
  content : list?
  type : string?
  script : (or/c path-string? (listof string?))
```

For backward compatibility. Like `make-element`, but adds `script-property` containing `type` and `script` to the element's style.

```
(struct with-attributes (style assoc)
  #:extra-constructor-name make-with-attributes)
  style : any/c
  assoc : (listof (cons/c symbol? string?))
```

For backward compatibility. Used for an `element`'s style to combine a base style with arbitrary HTML attributes. When the `style` field is itself an instance of `with-attributes`, its content is automatically flattened into the enclosing `with-attributes` when it is used (when, e.g., rendering an `element` or `paragraph`).

```
(struct target-url (addr style)
  #:extra-constructor-name make-target-url)
  addr : path-string?
  style : any/c
```

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

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

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

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

For backward compatibility. An alias for `content->string`.

4.8.2 Compatibility Basic Functions

```
(require scribble/basic)      package: scribble-lib
```

The `scribble/basic` compatibility library mostly just re-exports `scribble/base`.

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

For backward compatibility. Wraps the decoded `pre-content` as an element with style `style-name`.

```
(itemize itm ... [#:style style]) → itemization?
  itm : (or/c whitespace? an-item?)
  style : (or/c style? string? symbol? #f) = #f
```

For backward compatibility. Like `itemlist`, but whitespace strings among the `itms` are ignored.

5 Literate Programming

Programs written using `scribble/lp` are simultaneously two things: a program and a document describing the program.

Programs in `scribble/lp` are viewed in two different ways, either by running the program directly or by including it with `lp-include`. When running the program, all of the chunk expressions are collected and stitched together into a program, and the rest of the module is discarded. When using `lp-include`, the entire contents of the module are preserved and are treated like an ordinary Scribble document, where chunks are typeset in a manner similar to `codeblock`.

For example, consider this program:

```
#lang scribble/lp
```

Literate programs have chunks of code, like this one:

```
@chunk[<f>
  (define (f x)
    <fs-body>)]
```

and this one:

```
@chunk[<fs-body>
  (* x x)]
```

that, when assembled, produce a complete program, in this case:

```
@racketblock[(define (f x)
  (* x x))]
```

When this file is required in the normal manner, it defines a function `f` that squares its argument, and the documentation is ignored. When it is included with `lp-include`, it looks like this:

Literate programs have chunks of code, like this one:

```
<f> ::=
```

```
(define (f x)
  <fs-body>)
```


and this one:

```
<fs-body> ::=
```

```
( * x x )
```

that, when assembled, produce a complete program, in this case:

```
(define (f x)
  (* x x))
```

5.1 scribble/lp Language

```
#lang scribble/lp      package: scribble-lib
```

The `scribble/lp` language provides core support for literate programming.

```
(chunk id form ...)
```

Introduces a chunk, binding `id` for use in other chunks. Normally, `id` starts with `<` and ends with `>`.

When running a scribble program only the code inside the chunks is run; the rest is ignored.

If `id` is `<*>`, then this chunk is used as the main chunk in the file. If `<*>` is never used, then the first chunk in the file is treated as the main chunk. If some chunk is not referenced from the main chunk (possibly indirectly via other chunks that the main chunk references), then it is not included in the program and thus is not run.

```
(CHUNK id form ...)
```

Like `chunk`, but allows the use of `unsyntax` in the code part. If you want to use `unsyntax` to escape to Scribble, use `UNSYNTAX`.

5.2 scribble/lp-include Module

```
(require scribble/lp-include)  package: scribble-lib
```

The `scribble/lp-include` library is normally used within a Scribble document—that is, a module that starts with something like `#lang scribble/base` or `#lang scribble/manual`, instead of `#lang racket`.

| `(lp-include filename)`

Includes the source of *filename* as the typeset version of the literate program.

6 Low-Level Scribble API

6.1 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.

6.1.1 Typical Composition

A Scribble document normally starts

```
#lang scribble/manual
```

but it could also start

```
#lang scribble/base
```

or

```
#lang scribble/doc
```

The last one introduces the smallest number of typesetting bindings in the document body. Using `scribble/base` after `#lang` is the same as using `scribble/doc` plus `(require scribble/base)`, and using `scribble/manual` after `#lang` is the same as using `scribble/doc` plus `(require scribble/manual)`.

Besides making the file a module, each of the `#lang` declarations selects the Scribble reader (instead of the usual Racket 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> racket/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 `racket`.

6.1.2 Layer Roadmap

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

- `scribble/reader`: A reader that extends the syntax of Racket with `@`-forms for conveniently embedding a mixin of text and escapes. See §2 “@ Syntax”.
- `scribble/core`: 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 §6.3 “Structures And Processing”.

- `scribble/base-render` with `scribble/html-render`, `scribble/latex-render`, or `scribble/text-render`: A base renderer and mixins that generate documents in various formats from instances of the `scribble/struct` datatypes. See §6.4 “Renderers”.
- `scribble/decode`: Processes a stream of text, section-start markers, etc. to produce instances of the `scribble/core` datatypes. See §6.5 “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 `racket/base`. See §6.6 “Document Language”.
- `scribble/doc`: A language that combines `scribble/reader` with `scribble/doclang`. See §6.7 “Document Reader”.
- `scribble/base`: A library of basic document operators—such as `title`, `section`, and `secref`—for use with `scribble/decode` and a renderer. This library name also can be used as a language, where it combines `scribble/doc` with the exports of `scribble/base`. See §3.1 “Base Document Format”.
- `scribble/racket`: A library of functions for typesetting Racket code. See §4.4 “Racket”. These functions are not normally used directly, but instead used through `scribble/manual`.
- `scribble/manual`: A library of functions for writing Racket documentation; re-exports `scribble/base`. Also, the `scribble/manual-struct` library provides types for index-entry descriptions created by functions in `scribble/manual`. See §4.3 “Manual Forms”.
- `scribble/eval`: A library of functions for evaluating code at document-build time, especially for showing examples. See §4.5 “Evaluation and Examples”.
- `scribble/bnf`: A library of support functions for writing grammars. See §4.7 “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 §6.8 “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.

6.2 @ Reader Internals

6.2.1 Using the @ Reader

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

```
#reader scribble/reader @foo{This is free-form text!}
```

or use the `at-exp` meta-language as described in §6.2.3 “Adding @-expressions to a Language”.

Note that the Scribble reader reads @-forms as S-expressions. This means that it is up to you to give meanings for these expressions in the usual way: use Racket functions, define your functions, or require functions. For example, typing the above into racket is likely going to produce a “reference to undefined identifier” error, unless `foo` is defined. You can use `string-append` instead, or you can define `foo` as a function (with variable arity).

A common use of the Scribble @-reader is when using Scribble as a documentation system for producing manuals. In this case, the manual text is likely to start with

```
#lang scribble/doc
```

which installs the @ reader starting in “text mode,” wraps the file content afterward into a Racket module where many useful Racket and documentation related functions are available, and parses the body into a document using `scribble/decode`. See §6.7 “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:

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

6.2.2 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))])
                    (list fst prop rst)))))))]))
```


Implements the Scribble reader using the readtable produced by

```
(make-at-readtable #:command-readtable 'dynamic
                  #:datum-readtable 'dynamic)
```

Changed in version 1.1 of package `at-exp-lib`: Changed to use `'dynamic` for the command and datum readtables.

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

Like `read` and `read-syntax`, but starting as if inside a `@{...}` to return a (syntactic) list, which is useful for implementing languages that are textual by default.

The given `command-char` is used to customize the readtable used by the reader, effectively passing it along to `make-at-readtable`.

Changed in version 1.1 of package `at-exp-lib`: Changed to use `'dynamic` for the command and datum readtables.

```
(make-at-readtable [#:readtable readtable
                  #:command-char command-char
                  #:command-readtable command-readtable
                  #:datum-readtable datum-readtable
                  #:syntax-post-processor syntax-post-proc])
→ readtable?
  readtable : readtable? = (current-readtable)
  command-char : char? = #\@
  command-readtable : (or/c readtable? 'dynamic) = readtable
                    (or/c readtable?
                    boolean?
                    (readtable? . -> . readtable?)) = #t
                    'dynamic)
  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.
- `command-readtable` — determines the readtable that is extended for reading the command part of an @-form:
 - a readtable — extended to make `|` a delimiter instead of a symbol-quoting character
 - `'dynamic` — extends (`current-readtable`) at the point where a command is parsed to make `|` a delimiter
- `datum-readtable` — the readtable used for reading the datum part of an @-form:
 - `#t` — uses the constructed @-readtable itself
 - a readtable — uses the given readtable
 - a readtable-to-readtable function — called to construct a readtable from the generated @-readtable
 - `'dynamic` — uses (`current-readtable`) at the point where the datum part is parsed

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-proc` — 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")]))))
```

Changed in version 1.1 of package `at-exp-lib`: Added `#:command-readtable` and the `'dynamic` option for `#:datum-readtable`.

```
(make-at-reader #:syntax? syntax?
                #:inside? inside? ...) → procedure?
syntax? : #t
inside? : #f
```

Constructs a variant of a @-readtable. The arguments are the same as in `make-at-readtable`, with two more that determine the kind of reader function that will be created: `syntax?` chooses between a `read-` or `read-syntax-` like function, and `inside?` chooses a plain reader or an `-inside` variant.

The resulting function has a different contract and action based on these inputs. The expected inputs are as in `read` or `read-syntax` depending on `syntax?`; the function will read a single expression or, if `inside?` is true, the whole input; it will return a syntactic list of expressions rather than a single one in this case.

Note that `syntax?` defaults to `#t`, as this is the more expected common case when you're dealing with concrete-syntax reading.

Note that if `syntax?` is true, the `read`-like function is constructed by simply converting a syntax result back into a datum.

```
(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!`.

This is mostly useful for playing with the Scribble syntax on the REPL.

6.3 Structures And Processing

```
(require scribble/core)    package: scribble-lib
```

A document is represented as a part, as described in §6.3.1 “Parts, Flows, Blocks, and Paragraphs”. 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 four passes:

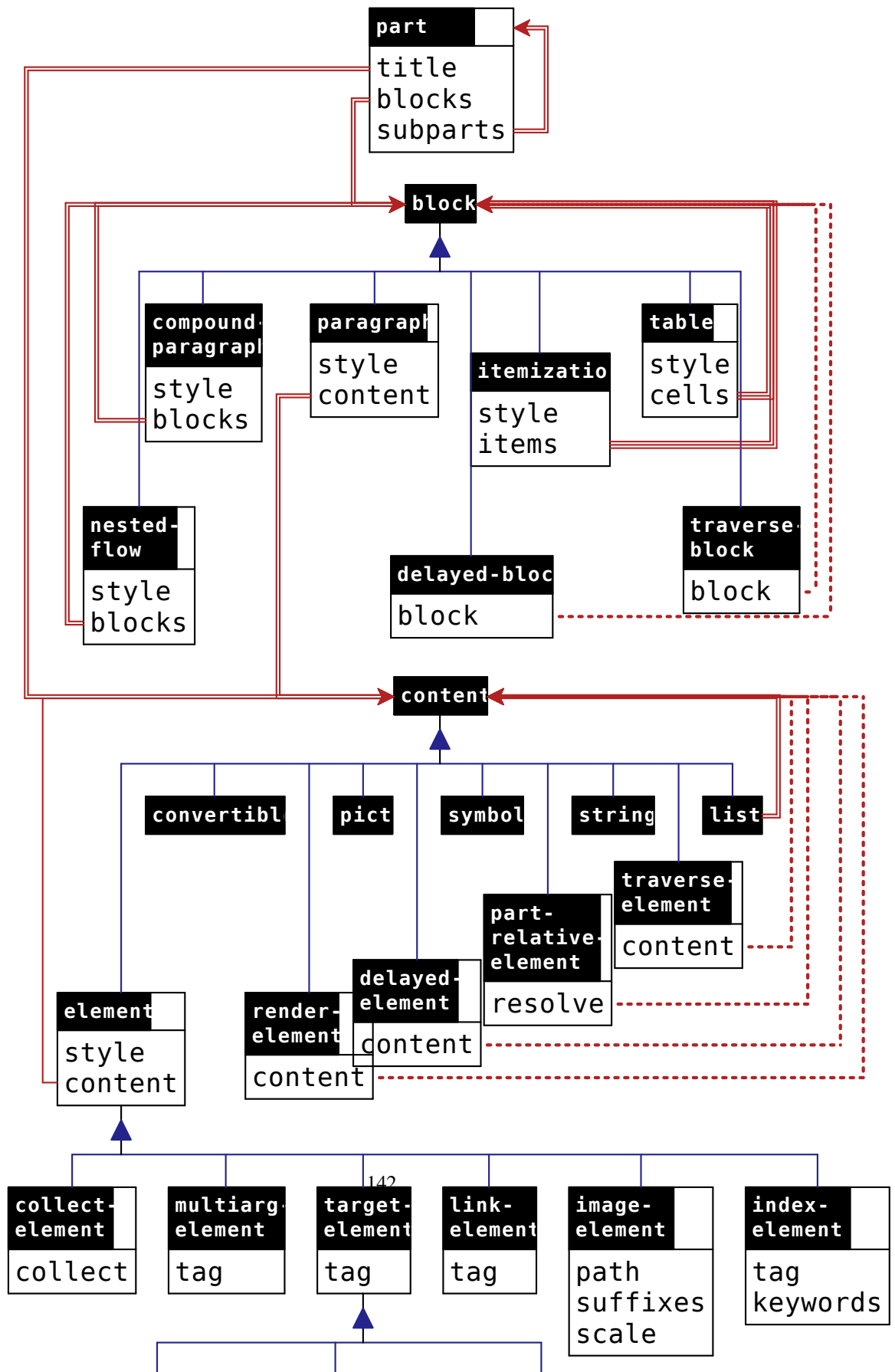
- The *traverse pass* traverses the document content in document order so that information from one part of a document can be communicated to other parts of the same document. The information is transmitted through a symbol-keyed mapping that can be inspected and extended by `traverse-elements` and `traverse-blocks` in the document. The traverse pass iterates the traversal until it obtains a fixed point (i.e., the mapping from one iteration is unchanged from the previous iteration).
- The *collect pass* globally collects information in the document that can span documents that are built at separate times, such as targets for hyperlinking.
- The *resolve pass* matches hyperlink references with targets and expands delayed elements (where the expansion should not contribute new hyperlink targets).
- The *render pass* generates the result document.

None of the passes mutate the document representation. Instead, the traverse pass, collect pass, and resolve pass accumulate information in a side hash table, `collect-info` table, and `resolve-info` table. The collect pass and resolve pass are effectively specialized version of traverse pass that work across separately built documents.

6.3.1 Parts, Flows, Blocks, and Paragraphs

This diagram shows the large-scale structure of the type hierarchy for Scribble documents. A box represents a struct or a built-in Racket type; for example `part` is a struct. The bottom portion of a box shows the fields; for example `part` has three fields, `title`, `blocks`, and `subparts`. The substruct relationship is shown vertically with navy blue lines connected by a triangle; for example, a `compound-paragraph` is a `block`. The types of values on fields are shown via dark red lines in the diagram. Doubled lines represent lists and tripled lines represent lists of lists; for example, the `blocks` field of `compound-paragraph` is a list of `blocks`. Dotted lines represent functions that compute elements of a given field; for example, the `block` field of a `traverse-block` struct is a function that computes a `block`.

The diagram is not completely accurate: a `table` may have `'cont` in place of a block in its `cells` field, and the types of fields are only shown if they are other structs in the diagram. A prose description with more detail follows the diagram.



A *part* is an instance of `part`; among other things, it has a title content, an initial flow, and a list of subsection parts. There is 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 a list of blocks.

A *block* is either a table, an itemization, a nested flow, a paragraph, a compound paragraph, a traverse block, or a delayed block.

- A *table* is an instance of `table`; it has a list of list of blocks corresponding to table cells.
- A *itemization* is an instance of `itemization`; it has a list of flows.
- A *nested flow* is an instance of `nested-flow`; it has a flow that is typeset as sub-flow.
- A *paragraph* is an instance of `paragraph`; it has a content:
 - An *content* can be a string, one of a few symbols, an instance of `element` (possibly `link-element`, etc.), a `multiarg-element`, a a traverse element, part-relative element, a delayed element, or a list of content.
 - * A string is included in the result document verbatim, except for space, and unless the content’s enclosing 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 content is either `'mdash`, `'ndash`, `'ldquo`, `'lsquo`, `'rdquo`, `'rsquo`, `'larr`, `'rarr`, or `'prime`; it is rendered as the corresponding HTML entity (even for Latex output).
 - * An instance of `element` has a content 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 contents to appear in the end-of-document index.
 - * An instance of `image-element` incorporates an image from a file into the rendered document.
 - * An instance of `multiarg-element` combines a style with a list of content, where the style corresponds to a rendered command that takes multiple arguments.

- * 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 *traverse element* is an instance of `traverse-element`, which ultimately produces content, but can accumulate and inspect information in the traverse pass.
 - * 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*. 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*.
 - * An instance of `render-element` has a procedure that is called in the render pass of document processing.
- A *compound paragraph* is an instance of `compound-paragraph`; like `blockquote`, it has list of blocks, but the blocks are typeset as a single paragraph (e.g., no indentation after the first block) instead of inset.
 - A *traverse block* is an instance of `traverse-block`, which ultimately produces another block, but can accumulate and inspect information during the traverse pass.
 - 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*.

6.3.2 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 Racket 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`, `'tech`, or `'cite`, or whose second item is a list that starts with `'prefixable`:

- The prefix is added to a string second item by creating a list containing the prefix and string.
- The prefix is added to a list second item after `'part`, `'tech`, or `'cite` using `cons`.
- The prefix is added to a second item that starts `'prefixable` by adding it to the list after `'prefixable`.
- A prefix is not added to a `generated-tag` item.

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 the prefix, 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.

The `scribble/tag` library provides functions for constructing tags.

6.3.3 Styles

A *style* combines a style name with a list of style properties in a `style` structure. A *style name* is either a string, symbol, or `#f`. A *style property* can be anything, including a symbol or a structure such as `color-property`.

A style has a single style name, because the name typically corresponds to a configurable instruction to a renderer. For example, with Latex output, a string style name corresponds to a Latex command or environment. For more information on how string style names interact with configuration of a renderer, see §6.11 “Extending and Configuring Scribble Output”. Symbolic style names, meanwhile, provide a simple layer of abstraction between the renderer and documents for widely supported style; for example, the `'italic` style name is supported by all renderers.

Style properties within a style compose with style names and other properties. Again, symbols are often used for properties that are directly supported by renderers. For example, `'unnumbered` style property for a part renders the part without a section number. Many properties are renderer-specific, such as a `hover-property` structure that associates text with an element to be shown in an HTML display when the mouse hovers over the text.

6.3.4 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 docu-

ments 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.

6.3.5 Structure Reference

```
(struct part (tag-prefix
              tags
              title-content
              style
              to-collect
              blocks
              parts)
  #:extra-constructor-name make-part)
tag-prefix : (or/c #f string?)
tags       : (listof tag?)
title-content : (or/c #f list?)
style      : style?
to-collect  : list?
blocks     : (listof block?)
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. Normally, `tags` should be a non-empty list, so that hyperlinks can target the section.

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

For the `style` field, the currently recognized symbolic style names are as follows:

- `'index` — The part represents an index.

The recognized style properties are as follows:

- `'unnumbered` — A section number is not computed or rendered for the section.
- `'hidden-number` — A section number is computed for the section, but it is not rendered as part of the section name.
- `'toc-hidden` — The part title is not shown in tables of contents, including in “on this page” boxes. For Latex rendering, the part title is omitted only if it is unnumbered or has a hidden number.
- `'hidden` — The part title is not shown; for Latex output, the part title is not shown only if its is empty, and in that case, it is also excluded from tables of contents. The `'toc-hidden` style property usually should be included with `'hidden` (for consistency in non-Latex output).
- `'grouper` — The part is numbered with a Roman numeral, and its subsections continue numbering as if they appeared in the preceding part. In other words, the part acts like a “part” in a book where chapter numbering is continuous across parts.
- `'toc` — Sub-parts of the part are rendered on separate pages for multi-page HTML mode.
- `'non-toc` — Initial sub-parts of the part are *not* rendered on separate pages for multi-page HTML mode; this style property applies only to the main part.
- `'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).
- `'quiet` — In HTML output and most other output modes, hides entries for sub-parts of this part in a `table-of-contents` or `local-table-of-contents` listing except when those sub-parts are top-level entries in the listing.
- `'no-toc` — As a style property for the main part of a rendered page, 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.
- `'no-sidebar` — As a style property for the main part of a document, causes the HTML output to not include an “on this page” margin box.
- `document-version` structure — A version number for this part and its sub-parts (except as overridden). When it is not "" may be used when rendering a document; at a minimum, a non-"" version is rendered when it is attached to a part representing the whole document. The default version for a document is (`version`). In rendered form, the version is normally prefixed with the word “Version,” but this formatting can be controlled by overriding `.version:before` and/or `.versionNoNav:before` in CSS for HTML rendering or by redefining the `\SVersionBefore` macro for Latex rendering (see §6.11 “Extending and Configuring Scribble Output”).

- `document-date` structure — A date for the part, normally used on a document’s main part for for Latex output. The default date for a document is `#f`, which avoids explicitly specifying a date at the Latex level, so that the current date is used as the document date. Set the date to `"` to suppress a date in an output document.
- `body-id` structure — Generated HTML uses the given string `id` attribute of the `<body>` tag; this style property can be set separately for parts that start different HTML pages, otherwise it is effectively inherited by sub-parts; the default is `"scribble-racket-lang.org"`, but `setup-plt` installs `"doc-racket-lang.org"` as the `id` for any document that it builds.
- `attributes` structure — Provides additional HTML attributes for the `<html>` tag when the part corresponds to its own HTML page.
- `head-extra` structure — Provides additional HTML content for the `<head>` tag when the part corresponds to its own HTML page.
- `color-property` structure — For HTML, applies a color to the part title.
- `background-color-property` structure — For HTML, applies a color to the background of the part title.
- `hover-property` structure — For HTML, adds a text label to the title to be shown when the mouse hovers over it.
- `render-convertible-as` structure — For HTML, controls how objects that subscribe to the `file/convertible` protocol are rendered.
- `document-source` structure — For HTML, provides a module path for the part’s source. Clicking on an HTML section title generated for the part or its sub-parts may show the module path plus a section-tag string, so that the user can create a reference to the section.

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 `blocks` field contains the part’s initial flow (before sub-parts).

The `parts` field contains sub-parts.

```
(struct paragraph (style content)
  #:extra-constructor-name make-paragraph)
style : style?
content : content?
```

A paragraph has a style and a content.

For the `style` field, a string style name corresponds to a CSS class for HTML output or a macro for Latex output (see §6.11.1 “Implementing Styles”). The following symbolic style names are recognized:

- `'author` — Typeset as the author of a document. Such paragraphs normally should appear only in the initial flow of a `part` for a document, where they are treated specially by the Latex renderer by moving the author information to the title.
- `'pretitle` — Typeset before the title of the enclosing part.
- `'wraps` — Like a `#f` style name, but not boxable in the sense of `box-mode` for Latex output.

When a paragraph's style is `#f`, then it is boxable in the sense of `box-mode` for Latex output.

The currently recognized style properties are as follows:

- `'omitable` — When a table cell contains a single `paragraph` with the `'omitable` style property, then when rendering to HTML, no `<p>` tag wraps the cell content.
- `'div` — Generates `<div>` HTML output instead of `<p>` (unless a `alt-tag` property is provided).
- `alt-tag` structure — Generates the indicated HTML tag instead of `<p>` or `<div>`.
- `attributes` structure — Provides additional HTML attributes for the `<p>`, `<div>`, or alternate tag.
- `body-id` structure — For HTML, uses the given string as an `id` attribute of the `<p>`, `<div>`, or alternate tag.
- `'never-indent`s — For Latex and compound paragraphs; see `compound-paragraph`.
- `box-mode` structure — For Latex output, uses an alternate rendering form for boxing contexts (such as a table cell); see `box-mode`.

```
(struct table (style blockss)
  #:extra-constructor-name make-table)
style : style?
blockss : (listof (listof (or/c block? 'cont)))
```

See also the `tabular` function.

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

Within `style`, a string style name corresponds to a CSS class for HTML output or an environment for Latex output (see §6.11.1 “Implementing Styles”). The following symbolic style names are also recognized:

- `'boxed` — Renders as a definition. This style name is not intended for use on a table that is nested within a `'boxed` table; nested uses may look right for some renders of the style but not others.
- `'centered` — Centers HTML output horizontally.
- `'block` — Prevents pages breaks in Latex output.

The following style properties are currently recognized:

- `table-columns` structure — Provides column-specific styles, but only `column-attributes` properties (if any) are used if a `table-cells` structure is included as a style property. See `table-cells` for information about how a column style is used for each cell.
- `table-cells` structure — Provides cell-specific styles. See `table-cells` for information about how the styles are used.
- `attributes` structure — Provides additional HTML attributes for the `<table>` tag.
- `body-id` structure — For HTML, uses the given string as an `id` attribute of the `<table>` tag.
- `'aux` — For HTML, include the table in the table-of-contents display for the enclosing part.
- `'never-indent`s — For Latex and compound paragraphs; see `compound-paragraph`.

For Latex output, a paragraph as a cell value is not automatically line-wrapped, unless a vertical alignment is specified for the cell through a `table-cells` or `table-columns` style property. To get a line-wrapped paragraph, use a `compound-paragraph` or use an element with a string style and define a corresponding Latex macro in terms of `\parbox`. For Latex output of blocks in the flow that are `nested-flows`, `itemizations`, `compound-paragraphs`, or `delayed-blocks`, the block is wrapped with `minipage` using `\linewidth` divided by the column count as the width.

```
(struct itemization (style blockss)
  #:extra-constructor-name make-itemization)
style : style?
blockss : (listof (listof block?))
```

A `itemization` has a `style` and a list of flows.

In `style`, a string style name corresponds to a CSS class for HTML output or a macro for Latex output (see §6.11.1 “Implementing Styles”). In addition, the following symbolic style names are recognized:

- `'compact` — Reduces space between items.
- `'ordered` — Generates `` HTML output instead of `` or an Latex enumeration instead of an itemization.

The following style properties are currently recognized:

- `attributes` structure — Provides additional HTML attributes for the `` or `` tag.
- `body-id` structure — For HTML, uses the given string as an `id` attribute of the `` or `` tag.
- `'never-indent`s — For Latex and compound paragraphs; see [compound-paragraph](#).

```
(struct nested-flow (style blocks)
  #:extra-constructor-name make-nested-flow)
style : any/c
blocks : (listof block?)
```

A nested flow has a style and a flow.

In `style`, the style name is normally a string that corresponds to a CSS class for HTML `<blockquote>` output or a Latex environment (see §6.11.1 “Implementing Styles”). The following symbolic style names are recognized:

- `'inset` — Insets the nested flow relative to surrounding text.
- `'code-inset` — Insets the nested flow relative to surrounding text in a way suitable for code. If the nested flow has a single block, then it is boxable in the sense of `box-mode` for Latex output.
- `'vertical-inset` — Insets the nested flow vertically relative to surrounding text, but not horizontally. If the nested flow has a single block, then it is boxable in the sense of `box-mode` for Latex output.

The following style properties are currently recognized:

- `'command` — For Latex output, a string style name is used as a command name instead of an environment name.
- `'multicommand` — For Latex output, a string style name is used as a command name with a separate argument for each block in `blocks`.

- `attributes` structure — Provides additional HTML attributes for the `<blockquote>` tag.
- `body-id` structure — For HTML, uses the given string as an `id` attribute of the `<blockquote>` tag.
- `'never-indent`s — For Latex and compound paragraphs; see `compound-paragraph`.
- `box-mode` structure — For Latex output, uses an alternate rendering form for boxing contexts (such as a table cell); see `box-mode`.
- `'decorative` — The content of the nested flow is intended for decoration. Text output skips a decorative nested flow.
- `alt-tag` structure — Generates the indicated HTML tag instead of `<blockquote>`.

```
(struct compound-paragraph (style blocks)
  #:extra-constructor-name make-compound-paragraph)
style : style?
blocks : (listof block?)
```

A compound paragraph has a style and a list of blocks.

For HTML, a `paragraph` block in `blocks` is rendered without a `<p>` tag, unless the paragraph has a style with a non-`#f` style name. For Latex, each block in `blocks` is rendered with a preceding `\noindent`, unless the block has the `'never-indent`s property (checking recursively in a `nested-flow` or `compound-paragraph` if the `nested-flow` or `compound-paragraph` itself has no `'never-indent`s property).

The `style` field of a compound paragraph is normally a string that corresponds to a CSS class for HTML output or Latex environment for Latex output (see §6.11.1 “Implementing Styles”). The following style properties are currently recognized:

- `'command` — For Latex output, a string style name is used as a command name instead of an environment name.
- `alt-tag` structure — Generates the given HTML tag instead of `<p>`.
- `attributes` structure — Provides additional HTML attributes for the `<p>` or alternate tag.
- `body-id` structure — For HTML, uses the given string as an `id` attribute of the `<p>` or alternate tag.
- `'never-indent`s — For Latex within another compound paragraph; see above.


```
(struct traverse-block (traverse)
  #:extra-constructor-name make-traverse-block)
traverse : block-traverse-procedure/c
```

Produces another block during the traverse pass, eventually.

The `traverse` procedure is called with `get` and `set` procedures to get and set symbol-keyed information; the `traverse` procedure should return either a block (which effectively takes the `traverse-block`'s place) or a procedure like `traverse` to be called in the next iteration of the traverse pass.

All `traverse-element` and `traverse-blocks` that have not been replaced are forced in document order relative to each other during an iteration of the traverse pass.

The `get` procedure passed to `traverse` takes a symbol and any value to act as a default; it returns information registered for the symbol or the given default if no value has been registered. The `set` procedure passed to `traverse` takes a symbol and a value to registered for the symbol.

See
also `cond-block` in
`scriblib/render-cond`.

The symbol `'scribble:current-render-mode` is automatically registered to a list of symbols that describe the target of document rendering. The list contains `'html` when rendering to HTML, `'latex` when rendering via Latex, and `'text` when rendering to text. The registration of `'scribble:current-render-mode` cannot be changed via `set`.

```
(struct delayed-block (resolve)
  #:extra-constructor-name make-delayed-block)
resolve : (any/c part? resolve-info? . -> . block?)
```

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)
  #:extra-constructor-name make-element)
style : element-style?
content : content?
```

Styled content within an enclosing paragraph or other content.

The `style` field can be a `style` structure, but it can also be just a style name.

In `style`, a string style name corresponds to a CSS class for HTML output and a macro name for Latex output (see §6.11.1 “Implementing Styles”). The following symbolic style names are recognized:

- `'tt`, `'italic`, `'bold`, `'roman`, `'sf`, `'url`, `'subscript`, `'superscript`, `'smaller`, `'larger` — Basic styles recognized by all renders.

- `'hspace` — Renders its `content` as monospace blanks.
- `'newline` — Renders a line break independent of the `content`.
- `'no-break` — Prevents line breaks when rendering `content`.

The following style properties are currently recognized:

- `target-url` structure — Generates a hyperlink.
- `url-anchor` structure — For HTML, inserts a hyperlink target before `content`.
- `color-property` structure — Applies a color to the text of `content`.
- `background-color-property` structure — Applies a color to the background of `content`.
- `alt-tag` structure — Generates the given HTML tag instead of the default one (``, `b`, etc.).
- `attributes` structure — Provides additional HTML attributes for a tag.
- `hover-property` structure — For HTML, adds a text label to the content to be shown when the mouse hovers over it.
- `script-property` structure — For HTML, supplies a script alternative to `content`.
- `body-id` structure — For HTML uses the given string as an `id` attribute of the `` tag.
- `'aux` — Intended for use in titles, where the auxiliary part of the title can be omitted in hyperlinks. See, for example, `secref`.
- `'tt-chars` — For Latex output, when the style name is a string, render the element's content with escapes suitable for Latex `tt` mode.
- `'exact-chars` — For Latex output, when the style name is a string or `#f`, render the elements content exactly (without escapes).
- `command-extras` structure — For Latex output, adds strings as arguments to the Latex command.

Changed in version 1.6 of package `scribble-lib`: Changed `'exact-chars` handling to take effect when the style name is `#f`.

```
(struct image-element element (path suffixes scale)
  #:extra-constructor-name make-image-element)
path : (or/c path-string?
        (cons/c 'collects (listof bytes?)))
suffixes : (listof #rx"^[.]")
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`.

For each string in `suffixes`, if the rendered works with the corresponding suffix, the suffix is added to `path` and used if the resulting path refers to a file that exists. The order in `suffixes` determines the order in which suffixes are tried. The HTML renderer supports ".png", ".gif", and ".svg", while the Latex renderer supports ".png", ".pdf", and ".ps" (but rendering Latex output to PDF will not work with ".ps" files, while rendering to Latex DVI output works only with ".ps" files). If `suffixes` is empty or if none of the suffixes lead to files that exist, `path` is used as-is.

The `scale` field scales the image in its rendered form.

```
(struct target-element element (tag)
  #:extra-constructor-name make-target-element)
tag : tag?
```

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

```
(struct toc-target-element target-element ()
  #:extra-constructor-name make-toc-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-target2-element toc-target-element (toc-content)
  #:extra-constructor-name make-toc-target2-element)
toc-content : content?
```

Extends `target-element` with a separate field for the content to be shown in the “on this page” table for HTML output.

```
(struct page-target-element target-element ()
  #:extra-constructor-name make-page-target-element)
```

Like `target-element`, but a link to the element goes to the top of the containing page.

```
(struct redirect-target-element target-element (alt-path
                                                alt-anchor)
  #:extra-constructor-name make-redirect-target-element)
alt-path : path-string?
alt-anchor : string?
```

Like `target-element`, but a link to the element is redirected to the given URL.

```
(struct toc-element element (toc-content)
  #:extra-constructor-name make-toc-element)
  toc-content : content?
```

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)
  #:extra-constructor-name make-link-element)
  tag : tag?
```

Hyperlinks the content to `tag`.

When `tag` is a part tag and the content of the element is `null`, then the hyperlink uses the target part’s number and/or title as the content. In that case, if the section number is preceded by a word, the word starts in uppercase if the element’s style includes a `'uppercase` property.

The following symbol is recognized as a style property:

- `'indirect-link` — For HTML output, treats the link as “external”. When rendering to HTML and the `set-external-tag-path` method is called to provide an external-link URL, then the resolution of the hyperlink can be deferred until the link is clicked (or, in some cases, patched by JavaScript when the documentation is viewed in a browser).

```
(struct index-element element (tag plain-seq entry-seq desc)
  #:extra-constructor-name make-index-element)
  tag : tag?
  plain-seq : (and/c pair? (listof string?))
  entry-seq : (listof content?)
  desc : any/c
```

The `plain-seq` specifies the keys for sorting, where the first string 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-string `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 multiarg-element (style contents)
  #:extra-constructor-name make-multiarg-element)
style : element-style?
contents : (listof content?)
```

Like `element` with a list for content, except that for Latex output, if the style name in `style` is a string, then it corresponds to a Latex command that accepts as many arguments (each in curly braces) as elements of `contents`.

```
(struct traverse-element (traverse)
  #:extra-constructor-name make-traverse-element)
traverse : element-traverse-procedure/c
```

Like `traverse-block`, but the `traverse` procedure must eventually produce content, rather than a block.

See also
`cond-element` in
`scribble/render-cond`.

```
(struct delayed-element (resolve sizer plain)
  #:extra-constructor-name make-delayed-element)
resolve : (any/c part? resolve-info? . -> . content?)
sizer : (-> any/c)
plain : (-> any/c)
```

The `render` procedure's arguments are the same as for `delayed-block`, but the result is content. 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 content 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 content when needed before the collect pass, such as when `element->string` is used before the collect pass.

```
(struct part-relative-element (resolve sizer plain)
  #:extra-constructor-name make-part-relative-element)
resolve : (collect-info? . -> . content?)
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)
  #:extra-constructor-name make-collect-element)
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 render-element element (render)
  #:extra-constructor-name make-render-element)
render : (any/c part? resolve-info? . -> . any)
```

Like `delayed-element`, but the `render` procedure is called during the render pass.

If a `render-element` instance is serialized (such as when saving collected info), it is reduced to a `element` instance.

```
(struct collected-info (number parent info)
  #:extra-constructor-name make-collected-info)
number : (listof (or/c #f exact-nonnegative-integer? string?))
parent : (or/c #f part?)
info : any/c
```

Computed for each part by the collect pass.

The length of the `number` list indicates the section's nesting depth. Numbers in `number` correspond to the section's number, its parent's number, etc. A non-empty string is used for a `'grouper` section. For an unnumbered section, `#f` is used in place of all numbers and `""` in place of all non-empty strings.

```
(struct target-url (addr)
  #:extra-constructor-name make-target-url)
addr : path-string?
```

Used as a style property for an `element`. A path is allowed for `addr`, but a string is interpreted as a URL rather than a file path.

```
(struct document-version (text)
  #:extra-constructor-name make-document-version)
text : (or/c string? #f)
```

Used as a style property for a [part](#) to indicate a version number.

```
(struct document-date (text)
  #:extra-constructor-name make-document-date)
text : (or/c string? #f)
```

Used as a style property for a [part](#) to indicate a date (which is typically used for Latex output).

```
(struct color-property (color)
  #:extra-constructor-name make-color-property)
color : (or/c string? (list/c byte? byte? byte?))
```

Used as a style property for an [element](#) to set its color. Recognized string names for `color` depend on the renderer, but at the recognized set includes at least "white", "black", "red", "green", "blue", "cyan", "magenta", and "yellow". When `color` is a list of bytes, the values are used as RGB levels.

When rendering to HTML, a `color-property` is also recognized for a block, [part](#) (and used for the title in the latter case) or cell in a [table](#).

```
(struct background-color-property (color)
  #:extra-constructor-name make-background-color-property)
color : (or/c string? (list/c byte? byte? byte?))
```

Like `color-property`, but sets the background color.

```
(struct table-cells (styleless)
  #:extra-constructor-name make-table-cells)
styleless : (listof (listof style?))
```

Used as a style property for a [table](#) to set its cells' styles.

If a cell style has a string name, it is used as an HTML class for the `<td>` tag or as a Latex command name.

The following are recognized as cell-style properties:

- `'left` — Left-align the cell content.
- `'right` — Right-align the cell content top baselines.
- `'center` — Center the cell content horizontally.
- `'top` — Top-align the cell content.

- `'baseline` — Align the cell content top baselines.
- `'bottom` — bottom-align the cell content.
- `'vcenter` — Center the cell content vertically.
- `'border` — Draw a line around all sides of the cell. Borders along a shared edge of adjacent cells are collapsed into a single line.
- `'left-border`, `'right-border`, `'top-border`, or `'bottom-border` — Draw a line along the corresponding side of the cell (with the same border collapsing as for `'border`).
- `color-property` structure — For HTML, applies a color to the cell content.
- `background-color-property` structure — For HTML, applies a color to the background of the cell.
- `attributes` — Provides additional HTML attributes for the cell's `<td>` tag.

Changed in version 1.1 of package `scribble-lib`: Added `color-property` and `background-color-property` support. Changed in version 1.4: Added `'border`, `'left-border`, `'right-border`, `'top-border`, and `'bottom-border` support.

```
(struct table-columns (styles)
  #:extra-constructor-name make-table-columns)
  styles : (listof style?)
```

Like `table-cells`, but with support for a `column-attributes` property in each style, and the `styles` list is otherwise duplicated for each row in the table. The non-`column-attributes` parts of a `table-columns` are used only when a `table-cells` property is not present along with the `table-columns` property.

For HTML table rendering, for each column that has a `column-attributes` property in the corresponding element of `styles`, the attributes are put into an HTML `col` tag within the table.

```
(struct box-mode (top-name center-name bottom-name)
  #:extra-constructor-name make-box-mode)
  top-name : string?
  center-name : string?
  bottom-name : string?
(box-mode* name) → box-mode?
  name : string?
```

As a style property, indicates that a nested flow or paragraph is *boxable* when it is used in a *boxing context* for Latex output, but a nested flow is boxable only if its content is also boxable.

A boxing context starts with a table cell in a multi-column table, and the content of a block in a boxing context is also in a boxing context. If the cell's content is boxable, then the content determines the width of the cell, otherwise a width is imposed. A paragraph with a `#f` style name is boxable as a single line; the `'wraps` style name makes the paragraph non-boxable so that its width is imposed and its content can use multiple lines. A table is boxable when that all of its cell content is boxable.

To generate output in box mode, the `box-mode` property supplies Latex macro names to apply to the nested flow or paragraph content. The `top-name` macro is used if the box's top line is to be aligned with other boxes, `center-name` if the box's center is to be aligned, and `bottom-name` if the box's bottom line is to be aligned. The `box-mode*` function creates a `box-mode` structure with the same name for all three fields.

A `box-mode` style property overrides any automatic boxed rendering (e.g., for a paragraph with style name `#f`). If a block has both a `box-mode` style property and a `'multicommand` style property, then the Latex macro `top-name`, `center-name`, or `bottom-name` is applied with a separate argument for each of its content.

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

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

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

Returns `#t` if `v` is a string, symbol, `element`, `multiarg-element`, `traverse-element`, `delayed-element`, `part-relative-element`, a convertible value in the sense of `convertible?`, or list of content. Otherwise, it returns `#f`.

```
(struct style (name properties)  
  #:extra-constructor-name make-style)  
  name : (or/c string? symbol? #f)  
  properties : list?
```

Represents a style.

```
plain : style?
```

A style (`make-style #f null`).

```
(element-style? v) → boolean?  
  v : any/c
```

Returns `#t` if `v` is a string, symbol, `#f`, or `style` structure.

```
(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 non-empty list of `serializable?` values.

```
(struct generated-tag ()  
  #:extra-constructor-name make-generated-tag)
```

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

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

Converts content 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).

```
(content-width c) → exact-nonnegative-integer?  
  c : content?
```

Returns the width in characters of the given content.

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

Returns the width in characters of the given block.

```
(struct collect-info (fp  
  ht  
  ext-ht  
  ext-demand  
  parts  
  tags  
  gen-prefix  
  relatives  
  parents)
```

```

#:extra-constructor-name make-collect-info)
fp : any/c
ht : any/c
ext-ht : any/c
ext-demand : (tag? collect-info? . -> . 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 searches)
 #:extra-constructor-name make-resolve-info)
ci : any/c
delays : any/c
undef : any/c
searches : 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? #f)
  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 §6.3.4 “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 `racket setup` is used to build documentation.

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

Like `render-get/ext?`, but the second result can be a string to indicate the source document’s identification as established via `load-xref` and a `#:doc-id` argument.

Added in version 1.1 of package `scribble-lib`.

```
(resolve-search dep-key p ri key) → void?
  dep-key : any/c
  p : (or/c part? #f)
  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 `racket setup` 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? #f)
  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 `racket` setup 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? #f)
  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.

```
(traverse-block-block b i) → block?
  b : traverse-block?
  i : (or/c resolve-info? collect-info?)
```

Produces the block that replaces `b`.

```
(traverse-element-content e i) → content?
  e : traverse-element?
  i : (or/c resolve-info? collect-info?)
```

Produces the content that replaces `e`.

```
block-traverse-procedure/c : contract?
```

Defined as

```
(recursive-contract
  ((symbol? any/c . -> . any/c)
   (symbol? any/c . -> . any)
   . -> . (or/c block-traverse-procedure/c
            block?)))
```

`element-traverse-procedure/c` : contract?

Defined as

```
(recursive-contract
  ((symbol? any/c . -> . any/c)
   (symbol? any/c . -> . any)
   . -> . (or/c element-traverse-procedure/c
            content?)))
```

6.3.6 HTML Style Properties

```
(require scribble/html-properties)
package: scribble-lib
```

The `scribble/html-properties` library provides datatypes used as style properties for HTML rendering.

```
(struct attributes (assoc)
  #:extra-constructor-name make-attributes)
assoc : (listof (cons/c symbol? string?))
```

Used as a style property to add arbitrary attributes to an HTML tag.

```
(struct alt-tag (name)
  #:extra-constructor-name make-alt-tag)
name : (and/c string? #rx"^[a-zA-Z0-9]+$")
```

Use as a style property for an `element`, `paragraph`, or `compound-paragraph` to substitute an alternate HTML tag (instead of ``, `<p>`, `div`, etc.).

```
(struct column-attributes (assoc)
  #:extra-constructor-name make-column-attributes)
assoc : (listof (cons/c symbol? string?))
```

Used as a style property on a style with `table-columns` to add arbitrary attributes to an HTML `col` tag within the table.

```
(struct url-anchor (name)
  #:extra-constructor-name make-url-anchor)
name : string?
```

Used as a style property with `element` to insert an anchor before the element.

```
(struct hover-property (text)
  #:extra-constructor-name make-hover-property)
text : string?
```

Used as a style property with `element` to add text that is shown when the mouse hovers over the element.

```
(struct script-property (type script)
  #:extra-constructor-name make-script-property)
type : string?
script : (or/c path-string? (listof string?))
```

Used as a style property with `element` to supply a script alternative to the element content.

```
(struct css-addition (path)
  #:extra-constructor-name make-css-addition)
  (or/c path-string?
    (cons/c 'collects (listof bytes?))
  path :
    url?
    bytes?)
```

Used as a style property to supply a CSS file (if `path` is a path, string, or list), URL (if `path` is a `url`) or content (if `path` is a byte string) to be referenced or included in the generated HTML. This property can be attached to any style, and all additions are collected to the top of the generated HTML page.

The `path` field can be a result of `path->main-collects-relative`.

```
(struct css-style-addition (path)
  #:extra-constructor-name make-css-style-addition)
  (or/c path-string?
    (cons/c 'collects (listof bytes?))
  path :
    url?
    bytes?)
```

Like `css-addition`, but added after any style files that are specified by a document and before any style files that are provided externally.

```
(struct js-addition (path)
  #:extra-constructor-name make-js-addition)
  (or/c path-string?
    path : (cons/c 'collects (listof bytes?))
           url?
           bytes?)
```

Like `css-addition`, but for a JavaScript file instead of a CSS file.

```
(struct js-style-addition (path)
  #:extra-constructor-name make-js-style-addition)
  (or/c path-string?
    path : (cons/c 'collects (listof bytes?))
           url?
           bytes?)
```

Like `css-style-addition`, but for a JavaScript file instead of a CSS file.

```
(struct body-id (value)
  #:extra-constructor-name make-body-id)
  value : string?
```

Used as a style property to associate an id attribute with an HTML tag.

```
(struct document-source (module-path)
  #:extra-constructor-name make-document-source)
  module-path : module-path?
```

Used as a style property to associate a module path with a part. Clicking on a section title within the part may show `module-path` with the part's tag string, so that authors of other documents can link to the section.

More specifically, the section title is given the HTML attributes `x-source-module` and `x-part-tag`. The `scribble/manual` style recognizes those tags to make clicking a title show cross-reference information.

Added in version 1.2 of package `scribble-lib`.

```
(struct html-defaults (prefix style extra-files)
  #:extra-constructor-name make-html-defaults)
  prefix : (or/c bytes? path-string?
            (cons/c 'collects (listof bytes?)))
  style : (or/c bytes? path-string?
           (cons/c 'collects (listof bytes?)))
  extra-files : (listof (or/c path-string?
                              (cons/c 'collects (listof bytes?))))
```


Like `latex-defaults`, but use for the scribble command-line tool's `--html` and `--htmls` modes.

```
(struct head-extra (xexpr)
  #:extra-constructor-name make-head-extra)
xexpr : xexpr/c
```

For a `part` that corresponds to an HTML page, adds content to the `<head>` tag.

```
(struct render-convertible-as (types)
  #:extra-constructor-name make-render-convertible-as)
types : (listof (or/c 'png-bytes 'svg-bytes))
```

For a `part` that corresponds to an HTML page, controls how objects that subscribe to the `file/convertible` protocol are rendered.

The alternatives in the `types` field are tried in order and the first one that succeeds is used in the html output.

```
(struct part-link-redirect (url)
  #:extra-constructor-name make-part-link-redirect)
url : url?
```

As a style property on a `part`, causes hyperlinks to the part to be redirected to `url` instead of the rendered part.

```
(struct link-resource (path)
  #:extra-constructor-name make-link-resource)
path : path-string?
```

As a style property on an `element`, causes the elements to be rendered as a hyperlink to (a copy of) `path`.

The file indicated by `path` is referenced in place when `render<%>` is instantiated with `refer-to-existing-files` as true. Otherwise, it is copied to the destination directory and potentially renamed to avoid conflicts.

```
(struct install-resource (path)
  #:extra-constructor-name make-install-resource)
path : path-string?
```

Like `link-resource`, but makes `path` accessible in the destination without rendering a hyperlink.

This style property is useful only when `render<%>` is instantiated with `refer-to-existing-files` as `#f`, and only when `path` does not match then name of any other file that is copied by the renderer to the destination.

6.3.7 Latex Style Properties

```
(require scribble/latex-properties)
package: scribble-lib
```

The `scribble/latex-properties` library provides datatypes used as style properties for Latex rendering.

```
(struct tex-addition (path)
  #:extra-constructor-name make-tex-addition)
  (or/c path-string?
  path : (cons/c 'collects (listof bytes?)
                bytes?))
```

Used as a style property to supply a ".tex" file (if `path` is a path, string, or list) or content (if `path` is a byte string) to be included in the generated Latex. This property can be attached to any style, and all additions are collected to the top of the generated Latex file.

The `path` field can be a result of `path->main-collects-relative`.

```
(struct latex-defaults (prefix style extra-files)
  #:extra-constructor-name make-latex-defaults)
  prefix : (or/c bytes? path-string?
            (cons/c 'collects (listof bytes?)))
  style : (or/c bytes? path-string?
           (cons/c 'collects (listof bytes?)))
  extra-files : (listof (or/c path-string?
                             (cons/c 'collects (listof bytes?))))
```

Used as a style property on the main `part` of a document to set a default prefix file, style file, and extra files (see §6.11.2 “Configuring Output”). The defaults are used by the `scribble` command-line tool for `and --latex` or `and --pdf` mode if none are supplied via `--prefix` and `--style` (where `extra-files` are used only when `prefix` is used). A byte-string value is used directly like file content, and a path can be a result of `path->main-collects-relative`.

Languages (used with `#lang`) like `scribble/manual` and `scribble/sigplan` add this property to a document to specify appropriate files for Latex rendering.

See also `scribble/latex-prefix`.

```
(struct command-extras (arguments)
  #:extra-constructor-name make-command-extras)
  arguments : (listof string?)
```

Used as a style property on an `element` to add extra arguments to the element’s command in Latex output.

6.4 Renderers

A renderer is an object that provides four main methods: `traverse`, `collect`, `resolve`, and `render`. Each method corresponds to a pass described in §6.3 “Structures And Processing”, and they are chained together by the `render` function to render a document.

6.4.1 Rendering Driver

```
(require scribble/render)      package: scribble-lib

(render docs
  names
  [#:render-mixin render-mixin
   #:dest-dir dest-dir
   #:helper-file-prefix helper-file-prefix
   #:prefix-file prefix-file
   #:style-file style-file
   #:style-extra-files style-extra-files
   #:extra-files extra-files
   #:image-preferences image-preferences
   #:xrefs xrefs
   #:info-in-files info-in-files
   #:info-out-file info-out-file
   #:redirect redirect
   #:redirect-main redirect-main
   #:directory-depth directory-depth
   #:quiet? quiet?
   #:warn-undefined? warn-undefined?]) → void?

docs : (listof part?)
names : (listof path-string?)
render-mixin : (class? . -> . class?) = render-mixin
dest-dir : (or/c #f path-string?) = #f
helper-file-prefix : (or/c #f string?) = #f
prefix-file : (or/c #f path-string?) = #f
style-file : (or/c #f path-string?) = #f
style-extra-files : (listof path-string?) = #f
extra-files : (listof path-string?) = #f
image-preferences : (listof (or/c 'ps 'pdf 'png 'svg 'gif))
                  = null
xrefs : (listof xref?) = null
info-in-files : (listof path-string?) = null
info-out-file : (or/c #f path-string?) = #f
redirect : (or/c #f string?) = #f
redirect-main : (or/c #f string?) = #f
```

```
directory-depth : exact-nonnegative-integer? = 0
quiet? : any/c = #t
warn-undefined? : any/c = (not quiet?)
```

Renders the given *docs*, each with an output name derived from the corresponding element of *names*. A directory path (if any) for a name in *names* is discarded, and the file suffix is replaced (if any) with a suitable suffix for the output format.

The *render-mixin* argument determines the output format. By default, it is *render-mixin* from *scribble/html-render*.

The *dest-dir* argument determines the output directory, which is created using *make-directory** if it is non-*#f* and does not exist already.

The *helper-file-prefix*, *prefix-file*, *style-file*, *style-extra-files*, and *extra-files* arguments are passed on to the *render%* constructor.

The *image-preferences* argument specified preferred formats for image files and conversion, where formats listed earlier in the list are more preferred. The renderer specified by *render-mixin* may not support all of the formats listed in *image-preferences*.

The *xrefs* argument provides extra cross-reference information to be used during the documents' resolve pass. The *info-in-files* arguments supply additional cross-reference information in serialized form. When the *info-out-file* argument is not *#f*, cross-reference information for the rendered documents is written in serialized form to the specified file.

The *redirect* and *redirect-main* arguments correspond to the *set-external-tag-path* and *set-external-root-url* methods of *render-mixin* from *scribble/html-render*, so they should be non-*#f* only for HTML rendering.

The *directory-depth* arguments correspond to the *set-directory-depth* method of *render-multi-mixin*.

If *quiet?* is a false value, output-file information is written to the current output port.

If *warn-undefined?* is a true value, then references to missing cross-reference targets trigger a warning message on the current error port.

Changed in version 1.4 of package *scribble-lib*: Added the *#:image-preferences* argument.

6.4.2 Base Renderer

```
(require scribble/base-render)    package: scribble-lib
```

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

`scribble/markdown-render`, or `scribble/html-render`, or `scribble/latex-render`.

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-render.rkt`" 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<%/> : interface?`

```
(send a-render traverse srcs dests) → (and/c hash? immutable?)
  srcs : (listof part?)
  dests : (listof path-string?)
```

Performs the traverse pass, producing a hash table that contains the replacements for `and` `traverse-blocks` and `traverse-elementss`. See `render` for information on the `dests` argument.

```
(send a-render collect srcs dests fp [demand]) → collect-
info?
  srcs : (listof part?)
  dests : (listof path-string?)
  fp : (and/c hash? immutable?)
  demand : (tag? collect-info? . -> . any/c)
           = (lambda (tag ci) #f)
```

Performs the collect pass. See `render` for information on the `dests` arguments. The `fp` argument is a result from the `traverse` method.

The `demand` argument supplies external tag mappings on demand. When the `collect-info` result is later used to find a mapping for a tag and no mapping is already available, `demand` is called with the tag and the `collect-info`. The `demand` function returns true to indicate when it adds information to the `collect-info` so that the lookup should be tried again; the `demand` function should return `#f` if it does not extend `collect-info`.

```
(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. The `ci` argument is a result from the `collect` method.

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

Produces the final output. The *ri* argument is a result from the `render` method. 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 serialize-infos ri count doc) → list?
  ri : resolve-info?
  count : exact-positive-integer?
  doc : part?
```

Like `serialize-info`, but produces *count* results that together have the same information as produced by `serialize-info`. The structure of *doc* is used to drive the partitioning (on the assumption that *ri* is derived from *doc*).

```
(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*.

```
(send a-render get-defined ci) → (listof tag?)
  ci : collect-info?
```

Returns a list of tags that were defined within the documents represented by *ci*.

```
(send a-render get-defineds ci count doc)
→ (listof (listof tag?))
  ci : collect-info?
  count : exact-positive-integer?
  doc : part?
```

Analogous to `serialize-infos`: returns a list of tags for each of `count` partitions of the result of `get-defined`, using the structure of `doc` to drive the partitioning.

```
(send a-render get-external ri) → (listof tag?)  
ri : resolve-info?
```

Returns a list of tags that were referenced but not defined within the documents represented by `ri` (though possibly found in cross-reference information transferred to `ri` via `xref-transfer-info`).

```
(send a-render get-undefined ri) → (listof tag?)  
ri : resolve-info?
```

Returns a list of tags that were referenced by the resolved documents with no target found either in the resolved documents represented by `ri` or cross-reference information transferred to `ri` via `xref-transfer-info`.

If multiple tags were referenced via `resolve-search` and a target was found for any of the tags using the same dependency key, then no tag in the set is included in the list of undefined tags.

```
render% : class?  
superclass: object%  
extends: render<%>
```

Represents a renderer.

```
(new render%  
  [dest-dir dest-dir]  
  [[refer-to-existing-files refer-to-existing-files]  
  [root-path root-path]  
  [prefix-file prefix-file]  
  [style-file style-file]  
  [style-extra-files style-extra-files]  
  [extra-files extra-files]  
  [image-preferences image-preferences]])  
→ (is-a?/c render%)  
dest-dir : path-string?  
refer-to-existing-files : any/c = #f  
root-path : (or/c path-string? #f) = #f  
prefix-file : (or/c path-string? #f) = #f  
style-file : (or/c path-string? #f) = #f  
style-extra-files : (listof path-string?) = null  
extra-files : (listof path-string?) = null
```

```
image-preferences : (listof (or/c 'ps 'pdf 'png 'svg 'gif))
                  = null
```

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 *refer-to-existing-files* is true, then when a document refers to external files, such as an image or a style file, then the file is referenced from its source location instead of copied to the document destination.

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 relative 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).

The *prefix-file*, *style-file*, and *style-extra-files* arguments set files that control output styles in a formal-specific way; see §6.11.2 “Configuring Output” for more information.

The *extra-files* argument names files to be copied to the output location, such as image files or extra configuration files.

The *image-preferences* argument specified preferred formats for image files and conversion, where formats listed earlier in the list are more preferred. The renderer may not support all of the formats listed in *image-preferences*.

Changed in version 1.4 of package *scribble-lib*: Added the *image-preferences* initialization argument.

6.4.3 Text Renderer

```
(require scribble/text-render)    package: scribble-lib
```

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

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

6.4.4 Markdown Renderer

```
(require scribble/markdown-render)
package: scribble-lib
```

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


argument extends/implements: `render<%>`

Specializes a `render<%>` class for generating Markdown text.

Code blocks are marked using the Github convention

```
```racket
```

so that they are lexed and formatted as Racket code.

### 6.4.5 HTML Renderer

```
(require scribble/html-render) package: scribble-lib
```

```
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 documents 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.

If the link is based on a cross-reference entry that has a document-identifying string (see `load-xref` and its `#:doc-id` argument), the document identifier is added as a `doc` query element, and a path to the target within the document is added as a `rel` query element.

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

Configures the renderer to redirect links to documents installed in the distribution's documentation directory to the given URL, using the URL as a replacement to the path of the distribution's document directory.

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

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

```
(send a-render-multi set-directory-depth depth) → void?
depth : exact-nonnegative-integer?
```

Sets the depth of directory structure used when rendering parts that are own their own pages. A value of 0 is treated the same as 1.

#### 6.4.6 Latex Renderer

```
(require scribble/latex-render) package: scribble-lib
```

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

Specializes a `render<%>` class for generating Latex input.

#### 6.4.7 PDF Renderer

```
(require scribble/pdf-render) package: scribble-lib
```

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

Specializes a `render<%>` class for generating PDF output via Latex, building on `render-mixin` from `scribble/latex-render`.

#### 6.4.8 Contract (Blue boxes) Renderer

```
(require scribble/contract-render)
package: scribble-lib
```

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

Overrides the `render` method of given renderer to record the content of the blue boxes (generated by `defproc`, `deform`, etc) that appear in the document.

```
(send an-override-render-mixin-multi render srcs
 dests
 ri) → void?

srcs : (listof part?)
dests : (listof path?)
ri : render-info?
```

Overrides `render` in `render<%>`.

In addition to doing whatever the super method does, also save the content of the blue boxes (rendered via a `scribble/text-render` renderer).

It saves this information in three pieces in a file inside the `dests` directories called "blueboxes.rktd". The first piece is a single line containing a (decimal, ASCII) number. That number is the number of bytes that the second piece of information occupies in the file. The second piece of information is a `hash` that maps `tag?` values to a list of offsets and line numbers that follow the hash table. For example, if the `hash` maps `'(def ((lib "x/main.rkt") abcdef))` to `'((10 . 3))`, then that means that the documentation for the `abcdef` export from the `x` collection starts 10 bytes after the end of the hash table and continues for 3 lines. Multiple elements in the list mean that that `tag?` has multiple blue boxes and each shows where one of the boxes appears in the file.

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

Just like `override-render-mixin-multi`, except it saves the resulting files in a different place.

```
(send an-override-render-mixin-single render srcs
 dests
 ri) → void?

srcs : (listof part?)
dests : (listof path?)
ri : render-info?
```

Overrides `render` in `render<%>`.

Just like `render`, except that it saves the file "blueboxes.rktd" in the same directory where each `dests` element resides.

## 6.5 Decoding Text

```
(require scribble/decode) package: scribble-lib
```

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 §4.8.1 “Compatibility Structures And Processing”).

At the flow level, decoding recognizes a blank line as a paragraph separator. Blocks and paragraphs without blank lines in between are collected into a compound paragraph.

At the content level, decoding makes just a few special text conversions:

- `---`: converted to `'mdash`
- `--`: 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: ’
- `‘`: converted to `'lsquo`, which is a fancy quote: ‘

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.

```
(pre-content? v) → boolean?
v : any/c
```

Returns `#t` if `v` is a *pre-content* value: a string or other non-list content, a list of pre-content values, or a `splice` containing a list of pre-content values; otherwise returns `#f`.

Pre-content is decoded into content by functions like `decode-content` and `decode-paragraph`.

```
(pre-flow? v) → boolean?
v : any/c
```

Returns `#t` if `v` is a *pre-flow* value: a string or other non-list content, a `block`, `#<void>`, a list of pre-flow values, or a `splice` containing a list of pre-flow values; otherwise returns `#f`.

Pre-flow is decoded into a flow (i.e., a list of blocks) by functions like `decode-flow`.

```
(pre-part? v) → boolean?
 v : any/c
```

Returns `#t` if `v` is a *pre-part* value: a string or other non-list content, a block, a `part`, a `title-decl`, a `part-start`, a `part-index-decl`, a `part-collect-decl`, a `part-tag-decl`, `#<void>`, a list of pre-part values, or a `splice` containing a list of pre-part values; otherwise returns `#f`.

A pre-part sequence is decoded into a `part` by functions like `decode` and `decode-part`.

```
(decode lst) → part?
 lst : (listof pre-part?)
```

Decodes a document, producing a part. In `lst`, lists and instances of `splice` are inlined into the list, and `#<void>`s are dropped. 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.

Portions of `lst` are within a part are decoded using `decode-flow`.

```
(decode-part lst tags title depth) → part?
 lst : (listof pre-part?)
 tags : (listof string?)
 title : (or/c #f list?)
 depth : exact-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 block?)
 lst : (listof pre-flow?)
```

Decodes a flow. In `lst`, lists and instances of `splice` are inlined into the list. A sequence of two or more newlines separated only by whitespace is parsed as a compound-paragraph separator.

Portions of `lst` are within a compound paragraph are decoded using `decode-compound-paragraph`.

```
(decode-compound-paragraph lst) → block?
 lst : (listof pre-flow?)
```

Decodes a compound paragraph. In *lst*, lists and instances of `splice` are inlined into the list. Instances of `paragraph` and other block datatypes are used as-is in the result. If the compound paragraph contains a single block, the block is returned without a `compound-paragraph` wrapper.

Portions of *lst* that are separated by blocks are decoded using `decode-content`.

```
(decode-paragraph lst) → paragraph?
 lst : (listof pre-content?)
```

Decodes a paragraph using `decode-content` to decode *lst* as the paragraph's content.

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

Decodes content. Elements at the start of the list that are whitespace (according to `whitespace?`) are dropped. Lists and splices in *lst* are flattened into the list, similarly dropping leading whitespace. Plain strings are decoded; non-string, non-list content is included in the result as-is.

```
(decode-elements lst) → list?
 lst : (listof pre-content?)
```

An alias for `decode-content`.

```
(decode-string s) → (listof content?)
 s : string?
```

Decodes a single string to produce content.

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

Returns `#t` if *v* is a string that contains only whitespace, `#f` otherwise.

```
(struct title-decl (tag-prefix tags version style content)
 #:extra-constructor-name make-title-decl)
 tag-prefix : (or/c #f string?)
 tags : (listof string?)
 version : (or/c string? #f)
 style : style?
 content : content?
```

See `decode` and `decode-part`. The `tag-prefix` and `style` fields are propagated to the resulting `part`. If the `version` field is not `#f`, it is propagated as a `document-version` style property on the `part`.

Dropping whitespace in nested lists and splices was a poor implementation choice that is left in place for compatibility. To protect against it, you can exploit the similarly unfortunate fact that an empty list does not count as whitespace.

```
(struct part-start (depth tag-prefix tags style title)
 #:extra-constructor-name make-part-start)
depth : integer?
tag-prefix : (or/c #f string?)
tags : (listof string?)
style : style?
title : content?
```

Like `title-decl`, but for a sub-part. See `decode` and `decode-part`.

```
(struct part-index-decl (plain-seq entry-seq)
 #:extra-constructor-name make-part-index-decl)
plain-seq : (listof string?)
entry-seq : list?
```

See `decode`. The two fields are as for `index-element`.

```
(struct part-collect-decl (element)
 #:extra-constructor-name make-part-collect-decl)
element : (or/c element? part-relative-element?)
```

See `decode`.

```
(struct part-tag-decl (tag)
 #:extra-constructor-name make-part-tag-decl)
tag : tag?
```

See `decode`.

```
(struct splice (run)
 #:extra-constructor-name make-splice)
run : list?
```

See `decode`, `decode-part`, and `decode-flow`.

```
(spliceof ctc) → flat-contract?
ctc : flat-contract?
```

Produces a contract for a `splice` instance whose run elements satisfy `ctc`.

```
(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.

## 6.6 Document Language

```
#lang scribble/doclang2 package: scribble-lib
```

The `scribble/doclang2` language provides everything from `racket/base`, except that it replaces the `#!/module-begin` form.

The `scribble/doclang2` `#!/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.

For example:

```
#lang racket
(module example scribble/doclang2
 "hello world, this is"
 " an example document")
(require 'example)
doc
```

The behavior of `scribble/doclang2` can be customized by providing `#:id`, `#:post-process`, and `#:exprs` arguments at the very beginning of the module.

- `#:id` names the top-level documentation binding. By default, this is `doc`.
- `#:post-process` processes the body of the module after `decode`. By default, this is `values`.
- `#:exprs` prepends an additional sequence of expressions to the beginning of the module's body. By default, this is the empty sequence `()`.

This example explicitly uses the defaults for all three keywords:

```
#lang racket
(module example scribble/doclang2
 #:id doc
 #:post-process values
 #:exprs ()
 "hello world, this is an example document")
(require 'example)
doc
```



The next toy example uses a different name for the documentation binding, and also adds an additional binding with a count of the parts in the document:

```
#lang racket
(module example scribble/doclang2
 #:id documentation
 #:post-process (lambda (decoded-doc)
 (set! number-of-parts (length (part-
parts decoded-doc))))
 decoded-doc)
 #:exprs ((title "My first expression!"))

 (require scribble/core
 scribble/base)

 (define number-of-parts #f)
 (provide number-of-parts)
 (section "part 1")
 "hello world"
 (section "part 2")
 "this is another document")

(require 'example)
number-of-parts
documentation
```

### 6.6.1 scribble/doclang

```
#lang scribble/doclang package: scribble-lib
```

The `scribble/doclang` language provides the same functionality as `scribble/doclang2`, where the configuration options are positional and mandatory. The first three elements in the `#!/module-begin`'s body must be the `id`, `post-process`, and `exprs` arguments.

Example:

```
#lang racket
(module* example scribble/doclang
 doc
 values
 ()
 (require scribble/base)
 (provide (all-defined-out)))
```

```

(define foo (para "hello again"))
"hello world, this is an example document"
(para "note the " (bold "structure")))

(module+ main
 (require (submod ".." example))
 (printf "I see doc is: ~s\n\n" doc)
 (printf "I see foo is: ~s" foo))

```

## 6.7 Document Reader

```
#lang scribble/doc package: at-exp-lib
```

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.

## 6.8 Cross-Reference Utilities

```
(require scribble/xref) package: scribble-lib
```

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
 [#:demand-source demand-source
 #:render% using-render%
 #:root root-path
 #:doc-id doc-id-str]) → xref?
sources : (listof (-> (or/c any/c (-> list?))))
demand-source : (tag? -> (or/c (-> any/c) #f))
 = (lambda (tag) #f)
using-render% : (implementation?/c render<%>)
 = (render-mixin render%)
root-path : (or/c path-string? false/c) = #f
doc-id-str : (or/c path-string? false/c) = #f
```

Creates a cross-reference record given a list of functions, `sources`.

Let *source* be a function in *sources*. The *source* function normally returns serialized information, *info*, which was formerly obtained from *serialize-info* in *render<%>*. The result of *source* can optionally be another function, which is in turn responsible for returning a list of *infos*. Finally, each *info* can be either serialized information, a *#f* to be ignored, or a value produced by *make-data+root* or *make-data+root+doc-id*, from which *data* part is used as serialized information, the *root* part overrides *root-path* for deserialization, and the *doc-id* part (if any) overrides *doc-id-string* to identify the source document.

The *demand-source* function can effectively add a new source to *sources* in response to a search for information on the given tag. The *demand-source* function returns *#f* to indicate that no new sources satisfy the given tag.

Since the format of serialized information is specific to a rendering class, the optional *using-render%* argument accepts the relevant class. It defaults to HTML rendering, partly because HTML-format information is usable by other formats (including Latex/PDF and text).

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*, but a *make-data+root* result for any *info* supplies an alternate path for deserialization of the *info*'s *data*.

If *doc-id-str* is not *#f*, it identifies each cross-reference entry as originating from *doc-id-str*. This identification is used when a rendering link to the cross-reference entry as an external query; see the *set-external-tag-path* method of *render-mixin*.

Use *load-collections-xref* from *setup/xref* to get all cross-reference information for installed documentation.

Changed in version 1.1 of package *scribble-lib*: Added the *#:doc-id* argument.

```
(xref-binding->definition-tag xref
 binding
 mode) → (or/c tag? false/c)

xref : xref?
 (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)))

binding :
```

```
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
 [#:external-root-url root-url
 #:render% using-render%])
→ (or/c false/c path?)
 (or/c false/c string?)
xref : xref?
tag : tag?
root-url : (or/c string? #f) = #f
using-render% : (implementation?/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.

If *root-url* is provided, then references to documentation in the main installation are redirected to the given URL.

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% : (implemenation?/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)
 #:extra-constructor-name make-entry)
 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.

```
(data+root? v) → boolean?
 v : any/c
(make-data+root data root) → data+root?
 data : any/c
 root : (or/c #f path-string?)
```

A value constructed by [make-data+root](#) can be returned by a source procedure for [load-xref](#) to specify a path used for deserialization.

```
(data+root+doc-id? v) → boolean?
 v : any/c
(make-data+root+doc-id data root doc-id) → data+root+doc-id?
 data : any/c
 root : (or/c #f path-string?)
 doc-id : string?
```

Extends [make-data+root+doc-id](#) to support an document-identifying string (see [load-xref](#)).

Added in version 1.1 of package `scribble-lib`.

## 6.9 Tag Utilities

```
(require scribble/tag) package: scribble-lib
```

The `scribble/tag` library provides utilities for constructing cross-reference tags. The library is re-exported by `scribble/base`.

```
(make-section-tag name
 [#:doc doc-mod-path
 #:tag-prefixes tag-prefixes]) → tag?
name : string?
doc-mod-path : (or/c module-path? #f) = #f
tag-prefixes : (or/c #f (listof string?)) = #f
```

Forms a tag that refers to a section whose “tag” (as provided by the `#:tag` argument to `section`, for example) is `name`. If `doc-mod-path` is provided, the tag references a section in the document implemented by `doc-mod-path` from outside the document. Additional tag prefixes (for intermediate sections, typically) can be provided as `tag-prefixes`.

```
(make-module-language-tag lang) → tag?
lang : symbol?
```

Forms a tag that refers to a section that contains `defmodulelang` for the language `lang`.

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

Returns `#t` if `v` is a taglet, `#f` otherwise.

A *taglet* is a value that can be combined with a symbol via `list` to form a tag, but that is not a *generated-tag*. A taglet is therefore useful as a piece of a tag, and specifically as a piece of a tag that can gain a prefix (e.g., to refer to a section of a document from outside the document).

```
(doc-prefix mod-path taglet) → taglet?
mod-path : (or/c #f module-path?)
taglet : taglet?
(doc-prefix mod-path extra-prefixes taglet) → taglet?
mod-path : (or/c #f module-path?)
extra-prefixes : (or/c #f (listof taglet?))
taglet : taglet?
```

Converts part of a cross-reference tag that would work within a document implemented by `mod-path` to one that works from outside the document, assuming that `mod-path` is not `#f`. That is, `mod-path` is converted to a taglet and added as prefix to an existing *taglet*.

If `extra-prefixes` is provided, then its content is added as a extra prefix elements before the prefix for `mod-path` is added. A `#f` value for `extra-prefixes` is equivalent to `'()`.

If `mod-path` is `#f`, then *taglet* is returned without a prefix (except adding `extra-prefixes`, if provided).

```
(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`.

```
(module-path-index->taglet mpi) → taglet?
 mpi : module-path-index?
```

Converts a module path index to a taglet—a normalized encoding of the path as an S-expression—that is interned via `intern-taglet`.

The string form of the taglet is used as prefix in a tag to form cross-references into the document that is implemented by the module referenced by `mpi`.

```
(intern-taglet v) → any/c
 v : any/c
```

Returns a value that is `equal?` to `v`, where multiple calls to `intern-taglet` for `equal?` vs produce the same (i.e., `eq?`) value.

## 6.10 Blue Boxes Utilities

```
(require scribble/blueboxes) package: scribble-lib
```

The `scribble/blueboxes` provides access to the content of the “blue boxes” that describe some module’s export (but without any styling).

```
(fetch-blueboxes-strings tag
 [#:blueboxes-cache blueboxes-cache])
→ (or/c #f (non-empty-listof string?))
 tag : tag?
 blueboxes-cache : blueboxes-cache? = (make-blueboxes-cache)
```

Returns a list of strings that show the content of the blue box (without any styling information) for the documentation referenced by `tag`.

The first string in the list describes the export (e.g. `"procedure"` when `defproc` is used, or `"syntax"` when `defform` was used to document the export).

```
(make-blueboxes-cache populate?) → blueboxes-cache?
 populate? : boolean?
```



Constructs a new (mutable) blueboxes cache.

If `populate?` is `#f`, the cache is initially unpopulated, in which case it is filled in the first time the cache is passed to `fetch-blueboxes-attrs`. Otherwise, the cache is initially populated.

```
(blueboxes-cache? v) → boolean?
v : any/c
```

Determines if `v` is a blueboxes cache.

## 6.11 Extending and Configuring Scribble Output

Sometimes, Scribble’s primitives and built-in styles are insufficient to produce the output that you need. The cases in which you need to extend or configure Scribble fall into two groups:

- You may need to drop into the back-end “language” of CSS or Latex to create a specific output effect. For this kind of extension, you will mostly likely attach a `css-addition` or `tex-addition` style property to style, where the addition implements the style name. This kind of extension is described in §6.11.1 “Implementing Styles”.
- You may need to produce a document whose page layout is different from the Racket documentation style. For that kind of configuration, you can run the `scribble` command-line tool and supply flags like `--prefix` or `++style`, or you can associate a `html-defaults` or `latex-defaults` style property to the main document’s style. This kind of configuration is described in §6.11.2 “Configuring Output”.

### 6.11.1 Implementing Styles

When a string is used as a style in an `element`, a `multiarg-element`, `paragraph`, `table`, `itemization`, `nested-flow`, or `compound-paragraph`, it corresponds to a CSS class for HTML output or a Latex macro/environment for Latex output. In Latex output, the string is used as a command name for a `paragraph` and an environment name for a `table`, `itemization`, `nested-flow`, or `compound-paragraph`; if the style has a `'command` style property for a `nested-flow` or `compound-paragraph`, then the style name is used as a command instead of an environment; and if the style has a `'multicommand` style property for a `nested-flow`, then the style name is used as a command with multiple arguments. In addition, for an itemization, the style string is suffixed with `"Item"` and used as a CSS class or Latex macro name to use for the itemization’s items (in place of `\item` in the case of Latex).

To add a mapping from your own style name to a CSS configuration, add a `css-addition` structure instance to a style’s style property list. To map a style name to a Latex macro or

environment, add a `tex-addition` structure instance. A `css-addition` or `tex-addition` is normally associated with the style whose name is implemented by the addition, but it can also be added to the style for an enclosing part.

Scribble includes a number of predefined styles that are used by the exports of `scribble/base`. You can use them or redefine them. The styles are specified by `"scribble.css"` and `"scribble.tex"` in the `"scribble"` collection.

The styles used by `scribble/manual` are implemented by `"racket.css"` and `"racket.tex"` in the `"scribble"` collection. Other libraries, such as `scrib-lib/autobib`, similarly implement styles through files that are associated by `css-addition` and `tex-addition` style properties.

To avoid collisions with future additions to Scribble, start your style name with an uppercase letter that is not `S`. An uppercase letter helps to avoid collisions with macros defined by Latex packages, and future styles needed by `scribble/base` and `scribble/manual` will start with `S`.

For example, a Scribble document

```
#lang scribble/manual
@(require scribble/core
 scribble/html-properties
 scribble/latex-properties)

@(define inbox-style
 (make-style "InBox"
 (list (make-css-addition "inbox.css")
 (make-tex-addition "inbox.tex"))))

@title{Quantum Pet}

Do not open: @elem[#:style inbox-style]{Cat}
```

combined with an `"inbox.css"` that contains

```
.InBox {
 padding: 0.2em;
 border: 1px solid #000000;
}
```

and an `"inbox.tex"` that contains

```
\newcommand{\InBox}[1]{\fbox{#1}}
```

generates

## Quantum Pet

Do not open: 

Scribble documents can also embed specific html tags and attributes. For example, this Scribble document:

```
#lang scribble/base

@(require scribble/core
 scribble/html-properties)

@(define external-image
 (elem
 #:style
 (style #f
 (list (alt-tag "img")
 (attributes
 '((src . "http://racket-
lang.org/icon.png"))))))))

@external-image
```

renders as the the Racket logo at the url <http://racket-lang.org/logo.png> when producing html.

### 6.11.2 Configuring Output

The implementation of styles used by libraries depends to some degree on separately configurable parameters, and configuration is also possible by replacing style implementations. Latex output is more configurable in the former way, since a document class determines a set of page-layout and font properties that are used by other commands. The style-replacement kind of configuration corresponds to re-defining Latex macros or overriding CSS class attributes. When `setup-plt` builds PDF documentation, it uses both kinds of configuration to produce a standard layout for Racket manuals; that is, it selects a particular page layout, and it replaces some `racket/base` styles.

Two kinds of files implement the two kinds of configuration:

- A *prefix file* determines the DOCTYPE line for HTML output or the `\documentclass` configuration (and perhaps some addition package uses or other configurations) for Latex output.

The default prefix files are "scribble-prefix.html" and "scribble-prefix.tex" in the "scribble" collection.

- A *style file* refines the implementation of styles used in the document—typically just the “built-in” styles used by `scribble/base`.

The default style files, "`scribble-style.css`" and "`scribble-style.tex`" in the "`scribble`" collection, change no style implementations.

For a given configuration of output, typically a particular prefix file works with a particular style file. Some prefix or style files may be more reusable. For now, reading the default files is the best way to understand how they interact. A prefix and/or style file may also require extra accompanying files; for example, a prefix file for Latex mode may require a corresponding Latex class file. The default prefix and style files require no extra files.

When rendering a document through the `scribble` command-line tool, use flags to select a prefix file, style file, and additional accompanying files:

- Select the prefix file using the `--prefix` flag. (Selecting the prefix file also cancels the default list of accompanying files, if any.)
- Replace the style file using the `--style` flag. Add additional style definitions and re-definitions using the `++style` flag.
- Add additional accompanying files with `++extra`.

When using the `scribble` command-line utility, a document can declare its default style, prefix, and extra files through a `html-defaults` and/or `latex-defaults` style property. In particular, when using the `scribble` command-line tool to generate Latex or PDF a document whose main part is implemented with `#lang scribble/manual`, the result has the standard Racket manual configuration, because `scribble/manual` associates a `latex-defaults` style property with the exported document. The `scribble/sigplan` language similarly associates a default configuration with an exported document. As libraries imported with `require`, however, `scribble/manual` and `scribble/sigplan` simply implement new styles in a composable way.

Whether or not a document has a default prefix- and style-file configuration through a style property, the defaults can be overridden using `scribble` command-line flags. Furthermore, languages like `scribble/manual` and `scribble/sigplan` add a `html-defaults` and/or `latex-defaults` style property to a main-document part only if it does not already have such a property added through the `#:style` argument of `title`.

### 6.11.3 Base CSS Style Classes

The following renderings of "`demo.scrbl`" demonstrate all of the CSS style classes used by `scribble/base` forms and functions:

- *S1 All-Styles Document, Title in “H2”* shows the default style in a single-page rendering without a search box.
- *M1 All-Styles Document, Title in “H2”* shows the default style in a multi-page rendering without a search box.
- *S2 All-Styles Document, Title in “H2”* shows the current manual style’s adjustments in a single-page rendering with a search box.
- *M2 All-Styles Document, Title in “H2”* shows the current manual style’s adjustments in a multi-page rendering with a search box.

The style classes:

<code>maincolumn</code>	Outer wrapper for all content in the main column.
<code>main</code>	Inner wrapper for all content in the main column, including navigation bars.
<code>refpara</code>	Outer wrapper for right-hand <code>margin-note</code> notes.
<code>refparaleft</code>	Outer wrapper for left-hand <code>margin-note</code> notes.
<code>refelem</code>	Outer wrapper for right <code>margin-note*</code> notes.
<code>refelemleft</code>	Outer wrapper for left-hand <code>margin-note*</code> notes.
<code>refcolumn</code>	Middle wrapper for right-hand <code>margin-note</code> and <code>margin-note*</code> notes.
<code>refcolumnleft</code>	Middle wrapper for left-hand <code>margin-note</code> and <code>margin-note*</code> notes.
<code>refcontent</code>	Inner wrapper for <code>margin-note</code> and <code>margin-note*</code> notes.
<code>tocset</code>	Groups table-of-contents panels: main and “on this page.”
<code>tocview</code>	Wraps the main (multi-page mode) or only (single-page mode) table-of-contents panel.
<code>tocviewlist</code>	A hierarchical layer of content in a main table-of-contents panel.
<code>tocviewlisttopspace</code>	With <code>tocviewlist</code> for the first layer.
<code>tocviewtoggle</code>	The always-visible name of a layer.
<code>tocviewtitle</code>	With <code>tocviewtoggle</code> for the first layer.
<code>tocviewsublist</code>	An item in a layer that has multiple items and more items before and after.
<code>tocviewsublistonly</code>	An item in a single-item layer.
<code>tocviewsublisttop</code>	The first item in a multi-item layer.
<code>tocviewsublistbottom</code>	The last item in a multi-item layer.
<code>tocviewlink</code>	Inner wrapper for an item in a layer when linked to a different page.
<code>tocviewselflink</code>	Inner wrapper for every item in a layer when linked to the same page.
<code>tocsub</code>	Wraps the “on this page” (multi-page mode only) table-of-contents panel.
<code>tocsubtitle</code>	Wraps the words “on this page”.
<code>tocsublist</code>	Inner table for the “on this page” panel.
<code>tocsublinknumber</code>	Number for an entry in an “on this page” panel.
<code>tocsubseclink</code>	Title for a <i>section</i> entry in an “on this page” panel.
<code>tocsubnonseclink</code>	Title for a <i>non-section</i> entry in an “on this page” panel that has some section links.
<code>tocsublink</code>	Title for a <i>non-section</i> entry in an “on this page” panel that has no section links.
<code>toctoplink</code>	Top-level entry in an inline (not the panel) table of contents.

<code>toclink</code>	Nested entry in an inline (not the panel) table of contents.
<code>versionbox</code>	Outer wrapper for version
<code>version</code>	Inner wrapper for version in the case of search box and/or navigation.
<code>versionNoNav</code>	Inner wrapper for version in the case of no search box and navigation.
<code>SAuthorListBox</code>	Outer wrapper for the author list.
<code>SAuthorList</code>	Inner wrapper for the author list.
<code>author</code>	Wrapper for an individual author.
<code>navsettop</code>	Wraps the top navigation bar (in multi-page mode or when a search bar is present).
<code>navsetbottom</code>	Wraps the bottom navigation bar (in multi-page mode or when a search bar is present).
<code>navleft</code>	Wraps left-side elements within a navigation bar.
<code>navright</code>	Wraps right-side elements within a navigation bar.
<code>nonavigation</code>	Disabled links within a navigation bar.
<code>searchform</code>	Outer wrapper for a search box within the top navigation bar.
<code>searchbox</code>	Inner wrapper for a search box within the top navigation bar.
<code>nosearchform</code>	Takes the place of an absent search box within the top navigation bar.
<code>SSubSubSubSection</code>	Deeply nested subsection (below <code>&lt;h5&gt;</code> ).
<code>SIntrapara</code>	Used with <code>&lt;div&gt;</code> instead of <code>&lt;p&gt;</code> for a paragraph within a <a href="#">compound-paragraph</a> .
<code>SubFlow</code>	For a <a href="#">nested-flow</a> with no style name: no inset.
<code>SCodeFlow</code>	For a <a href="#">nested-flow</a> with the <code>'code-inset</code> style name: inset suitable for code.
<code>SVInsetFlow</code>	For a <a href="#">nested-flow</a> with the <code>'vertical-inset</code> style name: add space before and after suitable for code.
<code>SCentered</code>	For a <a href="#">nested-flow</a> created by <code>centered</code> : horizontally centered.
<code>boxed</code>	For a <a href="#">table</a> with the <code>'boxed</code> style name: as a definition box.
<code>compact</code>	For an <a href="#">itemlist</a> with the <code>'compact</code> style name.
<code>techoutside</code>	Outer wrapper for a technical-term reference.
<code>techinside</code>	Inner wrapper for a technical-term reference.
<code>indexlink</code>	For an entry in the index.
<code>stt</code>	Fixed-width text.
<code>sroman</code>	Serif text.
<code>ssanserif</code>	Sans serif text.
<code>slant</code>	Oblique (as opposed to italic) text.
<code>Smaller</code>	Smaller text (as created by <a href="#">smaller</a> ).
<code>Larger</code>	Smaller text (as created by <a href="#">larger</a> ).
<code>hspace</code>	For whitespace produced by <a href="#">hspace</a> .
<code>nobreak</code>	Disable link breaks.
<code>badlink</code>	Broken cross-reference.

plainlink                      Hyperlink without an underline.

In addition, the SIEHidden style class is built in to all Scribble HTML output to hide an element on Internet Explorer 6.

#### 6.11.4 Manual CSS Style Classes

The following renderings of "demo-manual.scrbl" demonstrate all of the CSS style classes used by `scribble/manual` forms and functions in addition to the base style classes.

- *S1 Manual All-Styles Document* shows the original style in a single-page rendering without a search box.
- *M1 Manual All-Styles Document* shows the original style in a multi-page rendering without a search box.
- *S2 Manual All-Styles Document* shows the current manual style's adjustments in a single-page rendering with a search box.
- *M2 Manual All-Styles Document* shows the current manual style's adjustments in a multi-page rendering with a search box.

The style classes:

RktSym	Identifiers with no <code>for-label</code> binding.
RktValLink	Identifier with <code>for-label</code> binding to a variable definition.
RktValDef	Definition site of a variable, normally combined with <code>RktValLink</code> .
RktStxLink	Identifier with <code>for-label</code> binding to a syntactic-form definition.
RktStxDef	Definition site of a syntactic form, normally combined with <code>RktStxLink</code> .
RktSymDef	Definition site of an identifier without binding (normally a mistake), combined with <code>RktSym</code> .
RktVar	Local variable or meta-variable.
RktRes	REPL result.
RktOut	Output written to the current output port.
RktErr	Output written to the current error port.
RktCmt	A comment in Racket code.
RktVal	A literal value in Racket code.
RktPn	Parentheses, keywords, and similar delimiters in Racket code.
RktRdr	Reader shorthands in Racket code, except for commas.
RktMeta	An unquoting comma in Racket code.
highlighted	Hilighlited code (via <code>code:highlight</code> in <code>racketblock</code> , for example).
RktIn	Foreground for literal characters written with <code>litchar</code> .
RktInBG	Background for literal characters written with <code>litchar</code> .
RktModLink	A module name linked to the module's definition.
RktMod	A module name (normally <code>RktModLink</code> , instead).
RktKw	A "keyword;" not normally used.

<code>RktOpt</code>	Brackets for optional arguments (in function definitions).
<code>RktBlk</code>	Wrapper for multi-line Racket code blocks.
<code>defmodule</code>	Module definition block.
<code>RpackageSpec</code>	Package specification within a module-definition block.
<code>RBoxed</code>	Definition block; always combined with <code>boxed</code> .
<code>together</code>	Table within a <code>together</code> grouping.
<code>RBackgroundLabel</code>	Wrapper for “procedure,” “syntax,” etc., backing in a definition box.
<code>RBackgroundLabelInner</code>	Wrapper within <code>RBackgroundLabel</code> .
<code>RForeground</code>	Wrapper for element to appear over a <code>RBackgroundLabel</code> .
<code>prototype</code>	Wrapper for a multi-line procedure-definition prototype.
<code>argcontract</code>	Wrapper for a multi-line argument contract and default value.
<code>specgrammar</code>	Wrapper for a grammar with a syntactic-form definition box.
<code>inherited</code>	Wrapper for a margin “inherited methods” table.
<code>inheritedlbl</code>	Wrapper for “Inherited methods:” and “from” labels.
<code>leftindent</code>	Left-indented block, such as form <code>specsubform</code> .
<code>insetpara</code>	Inset block.
<code>Rfilebox</code>	Wrapper for a file box (via <code>filebox</code> ),
<code>Rfiletitle</code>	Outer wrapper for a file box title.
<code>Rfilename</code>	Inner wrapper for a file box title.
<code>Rfilecontent</code>	Wrapper for file box content.
<code>SHistory</code>	Wrapper for <code>history</code> paragraphs.
<code>RBibliography</code>	Wrapper for a <code>bibliography</code> section.

### 6.11.5 Base Latex Macros

The “`scribble.tex`” Latex configuration includes several macros and environments that you can redefine to adjust the output style:

- `\preDoc` — called before the document content; the default does nothing, while the `scribble/manual` configuration enabled `\sloppy`.
- `\postDoc` — called after the document content; the default does nothing.
- `\sectionNewpage` — called before each top-level section starts; the default does nothing, while the `scribble/manual` configuration uses `\newpage` to start each chapter on a new page.



- `\SecRefLocal{}{}{}` — the first argument is a Latex label, the second argument is a section number, and the third argument is a section title. This macro is used by `secref` to reference a section (other than a document or top-level section within a document) that has a number and that is local to the current document. The default expands to `\SecRef`, passing along just the second and third arguments (so that the label is ignored).
- `\SecRef{}{}{}` — like `\SecRefLocal`, but used when the referenced section is in a different document, so that no label is available. The default shows “section” followed by the section number (ignoring the title). The `scribble/manual` redefinition of this macro shows “§”, the section number, and the title in quotes.
- `\ChapRefLocal{}{}{}` and `\ChapRef{}{}{}` — like `\SecRefLocal` and `\SecRef`, but for a top-level section within a document. The default implementation defers to `\SecRefLocal` or `\SecRef`.
- `\PartRefLocal{}{}{}` and `\PartRef{}{}{}` — like `\SecRefLocal` and `\SecRef`, but for a top-level section within a document whose part has the `'grouper` style property. The default `\PartRef` shows “part” followed by the section number (ignoring the title).
- `\BookRefLocal{}{}{}` and `\BookRef{}{}{}` — like `\SecRefLocal` and `\SecRef`, but for a document (as opposed to a section within the document). The default `\BookRef` implementation shows the title in italic.
- `\SecRefLocalUC{}{}{}` and `\SecRefUC{}{}{}` — like `\SecRefLocal` and `\SecRef`, but for `Secref`. The default `\SecRefUC` shows “Section” followed by the section number.
- `\ChapRefLocalUC{}{}{}` and `\ChapRefUC{}{}{}` — like `\ChapRefLocal` and `\ChapRef`, but for `Secref`. The default `\ChapRefUC` implementation defers to `\SecRefUC`.
- `\PartRefLocalUC{}{}{}` and `\PartRefUC{}{}{}` — like `\PartRefLocal` and `\PartRef`, but for `Secref`. The default `\PartRefUC` shows “Part” followed by the section number.
- `\BookRefLocalUC{}{}{}` and `\BookRefUC{}{}{}` — like `\BookRefLocal` and `\BookRef`, but for `Secref`. The default `\BookRefUC` defers to `\BookRef`.
- `\SecRefLocalUN{}{}{}`, `\SecRefUCUN{}{}{}`, `\SecRefLocalUCUN{}{}{}`, `\SecRefUN{}{}{}`, `\PartRefLocalUN{}{}{}`, `\PartRefUN{}{}{}`, `\PartRefLocalUCUN{}{}{}`, `\PartRefUCUN{}{}{}`, `\BookRefLocalUN{}{}{}`, `\BookRefUN{}{}{}`, `\BookRefLocalUCUN{}{}{}`, `\BookRefUCUN{}{}{}`, `\ChapRefLocalUN{}{}{}`, `\ChapRefUN{}{}{}`, `\ChapRefLocalUCUN{}{}{}`, and `\ChapRefUCUN{}{}{}` — like `\SecRefLocal`, etc., but in the case that a section/part/chapter number is unavailable. The default implementation of `\BookRefUN` uses `\BookRef` with an empty first argument. The default `\SecRefLocalUN` expands to its second argument in quotes followed by “on page” as a `\pageref` using the first argument, while the default `\SecRefUN` expands to its only argument in



## 7 Running scribble

The `scribble` command-line tool (also available as `raco scribble`) runs a Scribble document and renders it to a specific format. Select a format with one of the following flags, where the output name *fn* is by default the document source name without its file suffix:

- `--html` — a single HTML page "*fn*.html", plus CSS sources and needed image files; this mode is the default if no format is specified
- `--htmls` — multiple HTML pages (and associated files) in a "*fn*" directory, starting with "*fn*/index.html"
- `--html-tree <n>` — HTML pages in a directory tree up to *<n>* layers deep; a tree of depth 0 is equivalent to using `--html`, and a tree of depth 1 is equivalent to using `--htmls`
- `--latex` — LaTeX source "*fn*.tex", plus any needed additional files (such as non-standard class files) needed to run `latex` or `pdflatex`
- `--pdf` — PDF "*fn*.pdf" that is generated via `pdflatex`
- `--dvi` — PDF "*fn*.pdf" that is generated via `latex`, `dvips`, and `psstopdf`
- `--latex-section <n>` — LaTeX source "*fn*.tex" plus additional ".tex" files to be included in the enclosing document's preamble, where the enclosing document must use the UTF-8 input encoding and T1 font encoding; use 1 for *<n>* to make the rendered document a section, 2 for a subsection, etc.
- `--text` — plain text in a single file "*fn*.txt", with non-ASCII content encoded as UTF-8
- `--markdown` — Markdown text in a single file "*fn*.md", with non-ASCII content encoded as UTF-8

Use `--dest-name` to specify a *fn* other than the default name, but only when a single source file is provided. Use the `--dest` flag to specify a destination directory (for any number of source files). Use `--dest-base` to add a prefix to the name of each support file that is generated or copied to the destination.

After all flags, provide one or more document sources, where each source declares a module. The module should either have a `doc` submodule that exports `doc` as a `part`, or it should directly export `doc` as a `part`. (The submodule is tried first, and the main module is not directly loaded or evaluated if the submodule can be loaded on its own.)

When multiple documents are rendered at the same time, cross-reference information in one document is visible to the other documents. See §7.2 “Handling Cross-References” for information on references that cross documents that are built separately.

Changed in version 1.4: Added `--dvi`.

## 7.1 Extra and Format-Specific Files

Use the `--style` flag to specify a format-specific file to adjust the output style file for certain formats. For HTML (single-page or multi-page) output, the style file should be a CSS file that is applied after all other CSS files, and that may therefore override some style properties. For Latex (or PDF) output, the style file should be a `.tex` file that can redefine Latex commands. When a particular Scribble function needs particular CSS or Latex support, however, a better option is to use a `css-addition` or `tex-addition` style property so that the support is included automatically; see §6.11 “Extending and Configuring Scribble Output” for more information.

In rare cases, use the `--style` flag to specify a format-specific base style file. For HTML (single-page or multi-page) output, the style file should be a CSS file to substitute for `"scribble.css"` in the `"scribble"` collection. For Latex (or PDF) output, the style file should be a `.tex` file to substitute for `"scribble.tex"` in the `"scribble"` collection. The `--style` flag is rarely useful, because the content of `"scribble.css"` or `"scribble.tex"` is weakly specified; replacements must define all of the same styles, and the set of styles can change across versions of Racket.

Use `--prefix` to specify an alternate format-specific start of the output file. For HTML output, the starting file specifies the `DOCTYPE` declaration of each output HTML file as a substitute for `"scribble-prefix.html"` in the `"scribble"` collection. For Latex (or PDF) output (but not Latex-section output), the starting file specifies the `\documentclass` declaration and initial `\usepackage` declarations as a substitute for `"scribble-prefix.tex"` in the `"scribble"` collection. See also `html-defaults`, `latex-defaults`, and §6.11 “Extending and Configuring Scribble Output”.

For any output form, use the `++extra` flag to add a needed file to the build destination, such as an image file that is referenced in the generated output but not included via `image` (which copies the file automatically).

## 7.2 Handling Cross-References

Cross references within a document or documents rendered together are always resolved. When cross references span documents that are rendered separately, cross-reference information needs to be saved and loaded explicitly. Cross-reference information is format-specific, but HTML-format information is usable for Latex (or PDF) or text rendering.

A Racket installation includes HTML-format cross-reference information for all installed documentation. Each document’s information is in a separate file, so that loading all relevant files would be tedious. The `+m` or `++main-xref-in` flag loads cross-reference information for all installed documentation, so

```
scribble +m mine.scrbl
```

renders "mine.scrbl" to "mine.html" with cross-reference links to the Racket installation's documentation. (The "racket-index" package must be installed to use `+m/++main-xref-in`.)

The `++xref-in` flag loads cross-reference information by calling a specified module's function. The `setup/xref` module provides `load-collections-xref` to load cross-reference information for all installed documentation, and `+m` or `++main-xref-in` is just a shorthand for `++xref-in setup/xref load-collections-xref`.

The `--redirect-main` flag for HTML output redirects links to the local installation's documentation to a given URL, such as `http://docs.racket-lang.org/`. Beware that documentation links sometimes change (although Scribble generates HTML paths and anchors in a relatively stable way), so `http://download.racket-lang.org/docs/version/html/` may be more reliable when building with an installation for *version*. The `--redirect-main` flag is ignored for non-HTML output.

The `--redirect` flag is like `--redirect-main`, except that it builds on the given URL to indicate a cross-reference tag that is more stable than an HTML path and anchor (in case the documentation for a function changes sections, for example). No server currently exists to serve such tag requests, however.

For cross-references among documentation that is not part of the Racket installation, use `--info-out` to save information from a document build and use `++info-in` to load previously saved information. For example, if "c.scrbl" refers to information in "a.scrbl" and "b.scrbl", then

```
scribble --info-out a.sxref a.scrbl

scribble --info-out b.sxref b.scrbl

scribble ++info-in a.sxref ++info-in b.sxref c.scrbl
```

builds "c.html" with cross-reference links into "a.html" and "b.html".

### 7.3 Selecting an Image Format

Use the `++convert <fmt>` flag to select *<fmt>* as a preferred image format to use when rendering a document that includes values that can be converted to different image formats. The *<fmt>* argument can be `pdf`, `ps`, `png`, `svg`, or `gif`, but a renderer typically supports only a subset of those formats.

Use `++convert <fmt>` multiple times to specify multiple preferred formats, where a *<fmt>* earlier in the command line take precedence over *<fmt>*s specified later.

For example, to generate Latex sources with images in Encapsulated PostScript format (so that the result works with `latex` instead of `pdflatex`), combine `--latex` with `++convert`

ps. To generate HTML pages with images converted to SVG format instead of PNG format, combine `--html` with `++convert svg`.

Changed in version 1.4: Added `++convert` support.

## 7.4 Passing Command-Line Arguments to Documents

When `scribble` loads and renders a document module, by default it sets `current-command-line-arguments` to an empty vector. Use the `++arg` flag (any number of times) to add a string to `current-command-line-arguments`.

For example,

```
scribble ++arg --mode ++arg fast turtle.scrbl
```

causes (`current-command-line-arguments`) to return `'#("--mode" "fast")` while `"turtle.scrbl"` is loaded and rendered, which could affect the content that `"turtle.scrbl"` generates if it uses `current-command-line-arguments`.

Changed in version 1.1: Added the empty-vector default and `++arg` flag.

## **Index**































