Scribble as Preprocessor

Version 9.2.0.6

Matthew Flatt
and Eli Barzilay

July 4, 2026

The `scribble/text` and `scribble/html` languages act as "preprocessor" languages for generating text or HTML. These preprocessor languages use the same @ syntax as the main Scribble tool (see *Scribble: The Racket Documentation Tool*), but instead of working in terms of a document abstraction that can be rendered to text and HTML (and other formats), the preprocessor languages work in a way that is more specific to the target formats.

Contents

Index **39**

1 Text Generation

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

The `scribble/text` language provides everything from `racket/base`, `racket/promise`, `racket/list`, and `racket/string`, but with additions and a changed treatment of the module top level to make it suitable as for text generation or a preprocessor language:

- The language uses `read-syntax-inside` to read the body of the module, similar to §6.7 "Document Reader". This means that by default, all text is read in as Racket strings; and @-forms can be used to use Racket functions and expression escapes.
- Values of expressions are printed with a custom `output` function. This function displays most values in a similar way to `display`, except that it is more convenient for a textual output.

When `scribble/text` is used via `require` instead of `#lang`, then it does not change the printing of values, it does not include the bindings of `racket/base`, `include` is provided as `include/text`, and `begin` is provided as `begin/text`.

1.1 Writing Text Files

The combination of the two features makes text in files in the `scribble/text` language be read as strings, which get printed out when the module is `required`, for example, when a file is given as an argument to `racket`. (In these example the left part shows the source input, and the right part the printed result.)

```
#lang scribble/text
Programming languages should
be designed not by piling
feature on top of feature, but
blah blah blah.
```

→

```
Programming languages should
be designed not by piling
feature on top of feature, but
blah blah blah.
```

Using @-forms, we can define and use Racket functions.

```
#lang scribble/text
@(require racket/list)
@(define Foo "Preprocessing")
@(define (3x . x)
   ;; racket syntax here
   (add-between (list x x x) " "))
@Foo languages should
be designed not by piling
feature on top of feature, but
@3x{blah}.
```

→

```
Preprocessing languages should
be designed not by piling
feature on top of feature, but
blah blah blah.
```

As demonstrated in this case, the `output` function simply scans nested list structures recursively, which makes them convenient for function results. In addition, `output` prints most values similarly to `display` — notable exceptions are void and false values which cause no output to appear. This can be used for convenient conditional output.

```
#lang scribble/text
@(define (errors n)
   (list n
         " error"
         (and (not (= n 1)) "s")))
You have @errors[3] in your code,
I fixed @errors[1].
```

→

```
You have 3 errors in your code,
I fixed 1 error.
```

Using the scribble @-forms syntax, you can write functions more conveniently too.

```
#lang scribble/text
@(define (errors n)
   ;; note the use of `unless'
   @list{@n error@unless[(= n 1)]{s}})
You have @errors[3] in your code,
I fixed @errors[1].
```

→

```
You have 3 errors in your code,
I fixed 1 error.
```

Following the details of the scribble reader, you may notice that in these examples there are newline strings after each definition, yet they do not show in the output. To make it easier to write definitions, newlines after definitions and indentation spaces before them are ignored.

```
#lang scribble/text

@(define (plural n)
   (unless (= n 1) "s"))

@(define (errors n)
   @list{@n error@plural[n]})

You have @errors[3] in your code,
  @(define fixed 1)
I fixed @errors[fixed].
```

→

```
You have 3 errors in your code,
I fixed 1 error.
```

These end-of-line newline strings are not ignored when they follow other kinds of expressions, which may lead to redundant empty lines in the output.

```
#lang scribble/text
@(define (count n str)
   (for/list ([i (in-range 1 (add1 n))])
     @list{@i @str,@"\n"}))
Start...
@count[3]{Mississippi}
... and I'm done.
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,

... and I'm done.
```

There are several ways to avoid having such empty lines in your output. The simplest way is to arrange for the function call's form to end right before the next line begins, but this is often not too convenient. An alternative is to use a `@;` comment, which makes the scribble reader ignore everything that follows it up to and including the newline. (These methods can be applied to the line that precedes the function call too, but the results are likely to have what looks like erroneous indentation. More about this below.)

```
#lang scribble/text
@(define (count n str)
   (for/list ([i (in-range 1 (+ n 1))])
     @list{@i @str,@"\n"}))
Start...
@count[3]{Mississippi
}... done once.

Start again...
@count[3]{Massachusetts}@;
... and I'm done again.
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,
... done once.

Start again...
1 Massachusetts,
2 Massachusetts,
3 Massachusetts,
... and I'm done again.
```

A better approach is to generate newlines only when needed.

```
#lang scribble/text
@(require racket/list)
@(define (counts n str)
   (add-between
    (for/list ([i (in-range 1 (+ n 1))])
      @list{@i @str,})
    "\n"))
Start...
@counts[3]{Mississippi}
... and I'm done.
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,
... and I'm done.
```

In fact, this is common enough that the `scribble/text` language provides a convenient facility: `add-newlines` is a function that is similar to `add-between` using a newline string as the default separator, except that false and void values are filtered out before doing so.

```
#lang scribble/text
@(define (count n str)
   (add-newlines
    (for/list ([i (in-range 1 (+ n 1))])
      @list{@i @str,})))
Start...
@count[3]{Mississippi}
... and I'm done.
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,
... and I'm done.
```

```
#lang scribble/text
@(define (count n str)
   (add-newlines
    (for/list ([i (in-range 1 (+ n 1))])
      @(and (even? i) @list{@i @str,}))))
Start...
@count[6]{Mississippi}
... and I'm done.
```

→

```
Start...
2 Mississippi,
4 Mississippi,
6 Mississippi,
... and I'm done.
```

The separator can be set to any value.

```
#lang scribble/text
@(define (count n str)
   (add-newlines #:sep ",\n"
    (for/list ([i (in-range 1 (+ n 1))])
      @list{@i @str})))
Start...
@count[3]{Mississippi}.
... and I'm done.
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi.
... and I'm done.
```

1.2 Defining Functions and More

(Note: most of the tips in this section are applicable to any code that uses the Scribble @-forms syntax.)

Because the Scribble reader is uniform, you can use it in place of any expression where it is more convenient. (By convention, we use a plain S-expression syntax when we want a Racket expression escape, and an @-forms for expressions that render as text, which, in the `scribble/text` language, is any value-producing expression.) For example, you can use an @-forms for a function that you define.

```
#lang scribble/text
@(define @bold[text] @list{*@|text|*})
An @bold{important} note.
```

→

```
An *important* note.
```

This is not commonly done, since most functions that operate with text will need to accept a variable number of arguments. In fact, this leads to a common problem: what if we want to write a function that consumes a number of "text arguments" rathen than a single "rest-like" body? The common solution for this is to provide the separate text arguments in the S-expression part of an @-forms.

```
#lang scribble/text
@(define (choose 1st 2nd)
   @list{Either @1st, or @|2nd|@"."})
@(define who "us")
@choose[@list{you're with @who}
        @list{against @who}]
```

→

```
Either you're with us, or against us.
```

You can even use @-formss with a Racket quote or quasiquote as the "head" part to make it shorter, or use a macro to get grouping of sub-parts without dealing with quotes.

```
#lang scribble/text
@(define (choose 1st 2nd)
   @list{Either @1st, or @|2nd|@"."})
@(define who "us")
@choose[@list{you're with @who}
        @list{against @who}]
@(define-syntax-rule (compare (x ...) ...)
   (add-newlines
    (list (list "* " x ...) ...)))
Shopping list:
@compare[@{apples}
         @{oranges}
         @{@(* 2 3) bananas}]
```

→

```
Either you're with us, or against us.
Shopping list:
* apples
* oranges
* 6 bananas
```

Yet another solution is to look at the text values and split the input arguments based on a specific token. Using `match` can make it convenient — you can even specify the patterns with @-formss.

```
#lang scribble/text
@(require racket/match)
@(define (features . text)
   (match text
     [@list{@|1st|@...
            ---
            @|2nd|@...}
      @list{>> Pros <<
            @1st;
            >> Cons <<
            @|2nd|.}]))
@features{fast,
          reliable
          ---
          expensive,
          ugly}
```

→

```
>> Pros <<
fast,
reliable;
>> Cons <<
expensive,
ugly.
```

In particular, it is often convenient to split the input by lines, identified by delimiting `"\n"` strings. Since this can be useful, a `split-lines` function is provided.

```
#lang scribble/text
@(require racket/list)
@(define (features . text)
   (add-between (split-lines text)
                ", "))
@features{red
          fast
          reliable}.
```
→ red, fast, reliable.

Finally, the Scribble reader accepts *any* expression as the head part of an @-form — even an @ form. This makes it possible to get a number of text bodies by defining a curried function, where each step accepts any number of arguments. This, however, means that the number of body expressions must be fixed.

```
#lang scribble/text
@(define ((choose . 1st) . 2nd)
   @list{Either you're @1st, or @|2nd|.})
@(define who "me")
@@choose{with @who}{against @who}
```
→ Either you're with me, or against me.

1.3 Using Printouts

Because the text language simply displays each toplevel value as the file is run, it is possible to print text directly as part of the output.

```
#lang scribble/text
First
@display{Second}
Third
```
→
```
First
Second
Third
```

Taking this further, it is possible to write functions that output some text *instead* of returning values that represent the text.

```
#lang scribble/text
@(define (count n)
   (for ([i (in-range 1 (+ n 1))])
     (printf "~a Mississippi,\n" i)))
Start...
@count[3]@; avoid an empty line
... and I'm done.
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,
... and I'm done.
```

This can be used to produce a lot of output text, even infinite.

```
#lang scribble/text
@(define (count n)
   (printf "~a Mississippi,\n" n)
   (count (add1 n)))
Start...
@count[1]
this line is never printed!
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,
4 Mississippi,
5 Mississippi,
...
```

However, you should be careful not to mix returning values with printouts, as the results are rarely desirable.

```
#lang scribble/text
@list{1 @display{two} 3}
```

→

```
two1  3
```

Note that you don't need side-effects if you want infinite output. The `output` function iterates thunks and (composable) promises, so you can create a loop that is delayed in either form.

```
#lang scribble/text
@(define (count n)
   (cons @list{@n Mississippi,@"\n"}
         (lambda ()
           (count (add1 n)))))
Start...
@count[1]
this line is never printed!
```

→

```
Start...
1 Mississippi,
2 Mississippi,
3 Mississippi,
4 Mississippi,
5 Mississippi,
...
```

1.4 Indentation in Preprocessed output

An issue that can be very important in many text generation applications is the indentation of the output. This can be crucial in some cases, if you're generating code for an indentation-sensitive language (e.g., Haskell, Python, or C preprocessor directives). To get a better understanding of how the pieces interact, you may want to review how the Scribble reader section, but also remember that you can use quoted forms to see how some form is read.

```
#lang scribble/text
@(format "~s" '@list{
                a      →   (list "a" "\n" "   " "b" "\n" "c")
                  b
                c})
```

The Scribble reader ignores indentation spaces in its body. This is an intentional feature, since you usually do not want an expression to depend on its position in the source. But the question is whether we *can* render some output text with proper indentation. The `output` function achieves that by introducing `block`s. Just like a list, a `block` contains a list of elements, and when one is rendered, it is done in its own indentation level. When a newline is part of a `block`'s contents, it causes the following text to appear with indentation that corresponds to the column position at the beginning of the block.

In addition, lists are also rendered as blocks by default, so they can be used for the same purpose. In most cases, this makes the output appear "as intended" where lists are used for nested pieces of text — either from a literal `list` expression, or an expression that evaluates to a list, or when a list is passed on as a value; either as a toplevel expression, or as a nested value; either appearing after spaces, or after other output.

```
#lang scribble/text
foo @block{1                    foo 1
           2                        2
           3}          →            3
foo @list{4                     foo 4
          5                         5
          6}                        6
```

```
#lang scribble/text
@(define (code . text)
   @list{begin
           @text
         end})
@code{first
      second
      @code{
        third
        fourth}
      last}
```

→

```
begin
  first
  second
  begin
    third
    fourth
  end
  last
end
```

```
#lang scribble/text
@(define (enumerate . items)
   (add-newlines #:sep ";\n"
    (for/list ([i (in-naturals 1)]
               [item (in-list items)])
      @list{@|i|. @item})))
Todo: @enumerate[@list{Install Racket}
                 @list{Hack, hack, hack}
                 @list{Profit}].
```

→

```
Todo: 1. Install Racket;
      2. Hack, hack, hack;
      3. Profit.
```

There are, however, cases when you need more refined control over the output. The `scribble/text` language provides a few functions for such cases in addition to `block`. The `splice` function groups together a number of values but avoids introducing a new indentation context. Furthermore, lists are not always rendered as `block`s — instead, they are rendered as `splice`s when they are used inside one, so you essentially use `splice` to avoid the "indentation group" behavior, and `block` to restore it.

```
#lang scribble/text
@(define (blah . text)
   @splice{{
     blah(@block{@text});
   }})
start
  @splice{foo();
          loop:}
  @list{if (something) @blah{one,
                             two}}
end
```

→

```
start
  foo();
loop:
  if (something) {
    blah(one,
         two);
  }
end
```

The `disable-prefix` function disables all indentation printouts in its contents, including

the indentation before the body of the `disable-prefix` value itself. It is useful, for example, to print out CPP directives.

```
#lang scribble/text
@(define (((IFFOO . var) . expr1) . expr2)
   (define (array e1 e2)
     @list{[@e1,
            @e2]})
   @list{var @var;
         @disable-prefix{#ifdef FOO}
         @var = @array[expr1 expr2];
         @disable-prefix{#else}
         @var = @array[expr2 expr1];
         @disable-prefix{#endif}})

function blah(something, something_else) {
  @disable-prefix{#include "stuff.inc"}
  @@@IFFOO{i}{something}{something_else}
}
```

→

```
function blah(something, something_else) {
#include "stuff.inc"
  var i;
#ifdef FOO
  i = [something,
       something_else];
#else
  i = [something_else,
       something];
#endif
}
```

If there are values after a `disable-prefix` value on the same line, they *will* get indented to the goal column (unless the output is already beyond it).

```
#lang scribble/text
@(define (thunk name . body)
   @list{function @name() {
           @body
         }})
@(define (ifdef cond then else)
   @list{@disable-prefix{#}ifdef @cond
           @then
         @disable-prefix{#}else
           @else
         @disable-prefix{#}endif})

@thunk['do_stuff]{
  init();
  @ifdef["HAS_BLAH"
    @list{var x = blah();}
    @thunk['blah]{
      @ifdef["BLEHOS"
        @list{@disable-prefix{#}@;
                include <bleh.h>
              bleh();}
        @list{error("no bleh");}]
    }]
  more_stuff();
}
```

→

```
function do_stuff() {
  init();
# ifdef HAS_BLAH
    var x = blah();
# else
    function blah() {
#     ifdef BLEHOS
#       include <bleh.h>
        bleh();
#     else
        error("no bleh");
#     endif
    }
# endif
  more_stuff();
}
```

There are cases where each line should be prefixed with some string other than a plain indentation. The `add-prefix` function causes its contents to be printed using some given string prefix for every line. The prefix gets accumulated to an existing indentation, and indentation in the contents gets added to the prefix.

```
#lang scribble/text
@(define (comment . body)
   @add-prefix["// "]{@body})
@comment{add : int int -> string}
char *foo(int x, int y) {
  @comment{
    skeleton:
      allocate a string
      print the expression into it
      @comment{...more work...}
  }
  char *buf = malloc(@comment{FIXME!
                              This is bad}
                     100);
}
```

→

```
// add : int int -> string
char *foo(int x, int y) {
  // skeleton:
  //   allocate a string
  //   print the expression into it
  //   // ...more work...
  char *buf = malloc(// FIXME!
                     // This is bad
                     100);
}
```

When combining `add-prefix` and `disable-prefix` there is an additional value that can be useful: `flush`. This is a value that causes `output` to print the current indentation and prefix. This makes it possible to get the "ignored as a prefix" property of `disable-prefix` but only for a nested prefix.

```
#lang scribble/text
@(define (comment . text)
   (list flush
         @add-prefix[" *"]{
           @disable-prefix{/*} @text */}))
function foo(x) {
  @comment{blah
           more blah
           yet more blah}
  if (x < 0) {
    @comment{even more
             blah here
             @comment{even
                      nested}}
    do_stuff();
  }
}
```

→

```
function foo(x) {
  /* blah
   * more blah
   * yet more blah */
  if (x < 0) {
    /* even more
     * blah here
     * /* even
     *  * nested */ */
    do_stuff();
  }
}
```

1.5 Using External Files

Using additional files that contain code for your preprocessing is trivial: the source text is still source code in a module, so you can `require` additional files with utility functions.

```
#lang scribble/text
@(require "itemize.rkt")
Todo:
@itemize[@list{Hack some}
         @list{Sleep some}
         @list{Hack some
               more}]
```

→

```
Todo:
* Hack some
* Sleep some
* Hack some
  more
```

"itemize.rkt"

```
#lang racket
(provide itemize)
(define (itemize . items)
  (add-between (map (lambda (item)
                      (list "* " item))
                    items)
               "\n"))
```

Note that the `at-exp` language can often be useful here, since such files need to deal with texts. Using it, it is easy to include a lot of textual content.

```
#lang scribble/text
@(require "stuff.rkt")
Todo:
@itemize[@list{Hack some}
         @list{Sleep some}
         @list{Hack some
               more}]
@summary
```

"stuff.rkt"

```
#lang at-exp racket/base
(require racket/list)
(provide (all-defined-out))
(define (itemize . items)
  (add-between (map (lambda (item)
                      @list{* @item})
                    items)
               "\n"))
(define summary
  @list{If that's not enough,
        I don't know what is.})
```

→

```
Todo:
* Hack some
* Sleep some
* Hack some
  more
If that's not enough,
I don't know what is.
```

Of course, the extreme side of this will be to put all of your content in a plain Racket module, using @-formss for convenience. However, there is no need to use the text language in this case; instead, you can `(require scribble/text)`, which will get all of the bindings that are available in the `scribble/text` language. Using `output`, switching from a preprocessed files to a Racket file is very easy —- choosing one or the other depends on whether it is more convenient to write a text file with occasional Racket expressions or the other way.

```
#lang at-exp racket/base
(require scribble/text racket/list)
(define (itemize . items)
  (add-between (map (lambda (item)
                      @list{* @item})
                    items)
               "\n"))
(define summary
  @list{If that's not enough,
        I don't know what is.})
(output
 @list{
   Todo:
   @itemize[@list{Hack some}
            @list{Sleep some}
            @list{Hack some
                  more}]
   @summary
 })
```

→

```
Todo:
* Hack some
* Sleep some
* Hack some
  more
If that's not enough,
I don't know what is.
```

However, you might run into a case where it is desirable to include a mostly-text file from a `scribble/text` source file. It might be because you prefer to split the source text to several files, or because you need to use a template file that cannot have a `#lang` header (for example, an HTML template file that is the result of an external editor). In these cases, the `scribble/text` language provides an `include` form that includes a file in the preprocessor syntax (where the default parsing mode is text).

```
#lang scribble/text
@(require racket/list)
@(define (itemize . items)
   (list
    "<ul>"
    (add-between
     (map (lambda (item)
            @list{<li>@|item|</li>})
          items)
     "\n")
    "</ul>"))
@(define title "Todo")
@(define summary
   @list{If that's not enough,
         I don't know what is.})

@include["template.html"]
```

"template.html"

```
<html>
<head><title>@|title|</title></head>
<body>
  <h1>@|title|</h1>
  @itemize[@list{Hack some}
           @list{Sleep some}
           @list{Hack some
                 more}]
  <p><i>@|summary|</i></p>
</body>
</html>
```

→

```
<html>
<head><title>Todo</title></head>
<body>
  <h1>Todo</h1>
  <ul><li>Hack some</li>
      <li>Sleep some</li>
      <li>Hack some
          more</li></ul>
  <p><i>If that's not enough,
        I don't know what is.</i></p>
</body>
</html>
```

(Using `require` with a text file in the `scribble/text` language will not work as intended: the language will display the text is when the module is invoked, so the required file's contents will be printed before any of the requiring module's text does. If you find yourself in such a situation, it is better to switch to a Racket-with-@-expressions file as shown above.)

1.6 Text Generation Functions

`outputable/c` : `contract?`

A contract that (in principle) corresponds to value that can be output by `output`. Currently, however, this contract accepts all values (to avoid the cost of checking at every boundary).

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

```
(output v [port]) → void?
  v : outputable/c
  port : output-port? = (current-output-port)
```

Outputs values to *port* as follows for each kind of *v*:

- strings, byte strings, symbols, paths, keywords, numbers, and characters: converts the value to a string along the same lines as `display`, and then passes the string to the *current writer*, which is initially `write-string`
- `#<void>`, `#f`, or `null`: no output
- list: output depends on the current mode, which is initially splice mode:
  - *block mode*: each item in order, using the starting column as the *current indentation* (which starts out empty)
  - *splice mode*: outputs each item in order
- `(block v2 ...)`: outputs each *v2* in block mode.
- `(splice v2 ...)`: outputs each *v2* in splice mode.
- `(set-prefix pfx v2 ...)`: sets the *current prefix*, which is initially empty, to *pfx* while outputting each *v2*.
- `(add-prefix pfx v2 ...)`: sets the current prefix to by adding *pfx* while outputting each *v2*.
- `(disable-prefix v2 ...)`: sets the current prefix to empty while outputting each *v2*.
- `(restore-prefix v2 ...)`: rewinds the current prefix by one enclosing adjustments while outputting each *v2*.
- `flush`: outputs the current indentation and current prefix.
- `(with-writer writer v2 ...)`: sets the current writer to *writer* with outputting each *v2*.
- promise: outputs the result of `(force v)`
- box: outputs the result of `(unbox v)`
- procedure of 0 arguments: outputs the result of `(v)`

Any other kind of *v* triggers an exception.

```
(block v ...) → outputable/c
  v : outputable/c
```

Produces a value that outputs each *v* in block mode.

```
(splice v ...) → outputable/c
  v : outputable/c
```

Produces a value that outputs each *v* in splice mode.

```
(disable-prefix v ...) → outputable/c
  v : outputable/c
(restore-prefix v ...) → outputable/c
  v : outputable/c
(add-prefix pfx v ...) → outputable/c
  pfx : (or/c string? exact-nonnegative-integer?)
  v : outputable/c
(set-prefix pfx v ...) → outputable/c
  pfx : (or/c string? exact-nonnegative-integer?)
  v : outputable/c
```

Produces a value that outputs with an adjusted current prefix. An integer as a prefix is equivalent to a string with as many space characters.

```
flush : void?
```

A value that outputs as the current indentation plus current prefix.

```
(with-writer writer v ...) → outputable/c
  writer : (or/c (->* (string? output-port?)
                      (exact-nonnegative-integer?)
                      any/c)
                 #f)
  v : outputable/c
```

Produces a value that outputs with an adjusted current writer, where `#f` indicates `write-string`.

```
(add-newlines items [#:sep sep]) → list?
  items : list?
  sep : an/y = "\n"
```

Like `add-between`, but first removes `#f` and `#<void>` elements of *items*.

```
(split-lines items) → (listof list?)
  items : list?
```

Converts *items* to a list of lists, where consecutive non-"\n" values are kept together in a nested list, and "\n" values are dropped.

```
(include/text maybe-char path-spec)

maybe-char =
           | #:command-char command-char
```

Like include from racket/include, but reads the file at *path-spec* in the same way as for scribble/text. If *command-char* is supplied, then it replaces @ as the escape character.

The scribble/text language via #lang provides include/text as include.

```
(begin/text form ...)
```

Like begin, but the results of expression forms are collected into a list that is returned as the result of the begin/list form.

The scribble/text language via #lang provides begin/text as begin.

2 HTML Generation

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

The `scribble/html` language provides a way to generate HTML that is different from `scribble/base`. The `scribble/base` approach involves describing a document that can be rendered to HTML, Latex, or other formats. The `scribble/html` approach, in contrast, treats the document content as HTML format plus escapes.

Specifically, `scribble/html` is like `scribble/text`, but with the following changes:

- The `scribble/html/html`, `scribble/html/xml`, and `scribble/html/resource` are re-exported, in addition to `scribble/text`.
- Free identifiers that end with `:` are implicitly quoted as symbols.

When `scribble/html` is used via `require` instead of `#lang`, then it does not change the printing of values, and it does not include the bindings of `racket/base`.

The `scribble/html/resource`, `scribble/html/xml`, and `scribble/html/html` libraries provide forms for generating HTML as strings to be output in the same way as `scribble/text`.

2.1 Generating HTML Strings

`(require scribble/html/html)` package: `scribble-html-lib`

The `scribble/html/html` provides functions for HTML representations that render to string form via `output-xml`.

```
(doctype s) → procedure?
  s : (or/c string 'html 'xhtml)
```

Produces a value that XML-renders as a DOCTYPE declaration.

Examples:

```
> (output-xml (doctype "?"))
<!DOCTYPE ?>
> (output-xml (doctype 'html))
<!DOCTYPE html>
> (regexp-split #rx"\n|((?<=\") (?=\"))"
                (xml->string (doctype 'xhtml)))
```

```
'("<?xml version=\"1.0\" encoding=\"utf-8\"?>"
  "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\""
  "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">"
  "")
```

```
(xhtml content ...) → procedure?
  content : outputable/c
```

Produces a value that XML-renders as the given content wrapped as XHTML.

Example:

```
> (regexp-split #rx"\n|((?<=\") (?=\"))"
                (xml->string (xhtml "Hello")))
'("<?xml version=\"1.0\" encoding=\"utf-8\"?>"
  "<!DOCTYPE html PUBLIC \"-//W3C//DTD XHTML 1.0 Strict//EN\""
  "\"http://www.w3.org/TR/xhtml1/DTD/xhtml1-strict.dtd\">"
  "<html xmlns=\"http://www.w3.org/1999/xhtml\">Hello</html>"
  "")
```

```
(html v ...) → procedure?
  v : outputable/c
(head v ...) → procedure?
  v : outputable/c
(title v ...) → procedure?
  v : outputable/c
(style v ...) → procedure?
  v : outputable/c
(script v ...) → procedure?
  v : outputable/c
(noscript v ...) → procedure?
  v : outputable/c
(slot v ...) → procedure?
  v : outputable/c
(frameset v ...) → procedure?
  v : outputable/c
(frame v ...) → procedure?
  v : outputable/c
(iframe v ...) → procedure?
  v : outputable/c
(noframes v ...) → procedure?
  v : outputable/c
(body v ...) → procedure?
  v : outputable/c
(div v ...) → procedure?
  v : outputable/c
```

```
(p v ...) → procedure?
  v : outputable/c
(h1 v ...) → procedure?
  v : outputable/c
(h2 v ...) → procedure?
  v : outputable/c
(h3 v ...) → procedure?
  v : outputable/c
(h4 v ...) → procedure?
  v : outputable/c
(h5 v ...) → procedure?
  v : outputable/c
(h6 v ...) → procedure?
  v : outputable/c
(hgroup v ...) → procedure?
  v : outputable/c
(ul v ...) → procedure?
  v : outputable/c
(ol v ...) → procedure?
  v : outputable/c
(menu v ...) → procedure?
  v : outputable/c
(dir v ...) → procedure?
  v : outputable/c
(li v ...) → procedure?
  v : outputable/c
(dl v ...) → procedure?
  v : outputable/c
(dt v ...) → procedure?
  v : outputable/c
(dd v ...) → procedure?
  v : outputable/c
(address v ...) → procedure?
  v : outputable/c
(pre v ...) → procedure?
  v : outputable/c
(blockquote v ...) → procedure?
  v : outputable/c
(center v ...) → procedure?
  v : outputable/c
(ins v ...) → procedure?
  v : outputable/c
(del v ...) → procedure?
  v : outputable/c
(a v ...) → procedure?
  v : outputable/c
```

```
(span v ...) → procedure?
  v : outputable/c
(bdo v ...) → procedure?
  v : outputable/c
(em v ...) → procedure?
  v : outputable/c
(strong v ...) → procedure?
  v : outputable/c
(dfn v ...) → procedure?
  v : outputable/c
(code v ...) → procedure?
  v : outputable/c
(samp v ...) → procedure?
  v : outputable/c
(kbd v ...) → procedure?
  v : outputable/c
(var v ...) → procedure?
  v : outputable/c
(cite v ...) → procedure?
  v : outputable/c
(abbr v ...) → procedure?
  v : outputable/c
(acronym v ...) → procedure?
  v : outputable/c
(q v ...) → procedure?
  v : outputable/c
(sub v ...) → procedure?
  v : outputable/c
(sup v ...) → procedure?
  v : outputable/c
(tt v ...) → procedure?
  v : outputable/c
(i v ...) → procedure?
  v : outputable/c
(b v ...) → procedure?
  v : outputable/c
(big v ...) → procedure?
  v : outputable/c
(small v ...) → procedure?
  v : outputable/c
(u v ...) → procedure?
  v : outputable/c
(s v ...) → procedure?
  v : outputable/c
(strike v ...) → procedure?
  v : outputable/c
```

```
(font v ...) → procedure?
  v : outputable/c
(object v ...) → procedure?
  v : outputable/c
(applet v ...) → procedure?
  v : outputable/c
(form v ...) → procedure?
  v : outputable/c
(label v ...) → procedure?
  v : outputable/c
(select v ...) → procedure?
  v : outputable/c
(optgroup v ...) → procedure?
  v : outputable/c
(option v ...) → procedure?
  v : outputable/c
(textarea v ...) → procedure?
  v : outputable/c
(fieldset v ...) → procedure?
  v : outputable/c
(legend v ...) → procedure?
  v : outputable/c
(button v ...) → procedure?
  v : outputable/c
(table v ...) → procedure?
  v : outputable/c
(caption v ...) → procedure?
  v : outputable/c
(thead v ...) → procedure?
  v : outputable/c
(tfoot v ...) → procedure?
  v : outputable/c
(tbody v ...) → procedure?
  v : outputable/c
(colgroup v ...) → procedure?
  v : outputable/c
(tr v ...) → procedure?
  v : outputable/c
(th v ...) → procedure?
  v : outputable/c
(td v ...) → procedure?
  v : outputable/c
(details v ...) → procedure?
  v : outputable/c
(dialog v ...) → procedure?
  v : outputable/c
```

```
(menuitem v ...) → procedure?
  v : outputable/c
```

Like `element/not-empty`, but with the symbolic form of the function name added as the first argument.

Example:

```
> (output-xml (title "The Book"))
<title>The Book</title>
```

```
(base v ...) → procedure?
  v : outputable/c
(meta v ...) → procedure?
  v : outputable/c
(link v ...) → procedure?
  v : outputable/c
(hr v ...) → procedure?
  v : outputable/c
(br v ...) → procedure?
  v : outputable/c
(basefont v ...) → procedure?
  v : outputable/c
(param v ...) → procedure?
  v : outputable/c
(img v ...) → procedure?
  v : outputable/c
(area v ...) → procedure?
  v : outputable/c
(input v ...) → procedure?
  v : outputable/c
(isindex v ...) → procedure?
  v : outputable/c
(col v ...) → procedure?
  v : outputable/c
(embed v ...) → procedure?
  v : outputable/c
(keygen v ...) → procedure?
  v : outputable/c
(wbr v ...) → procedure?
  v : outputable/c
```

Like `element`, but with the symbolic form of the function name added as the first argument.

Example:

```
> (output-xml (hr))
<hr />
```

```
nbsp : procedure?
ndash : procedure?
mdash : procedure?
bull : procedure?
middot : procedure?
sdot : procedure?
lsquo : procedure?
rsquo : procedure?
sbquo : procedure?
ldquo : procedure?
rdquo : procedure?
bdquo : procedure?
lang : procedure?
rang : procedure?
dagger : procedure?
Dagger : procedure?
plusmn : procedure?
deg : procedure?
```

The result of `(entity 'id)` for each *id*.

Example:

```
> (output-xml nbsp)
 
```

```
(script/inline v ...) → procedure?
  v : outputable/c
```

Produces a value that renders as an inline script.

Example:

```
> (output-xml (script/inline type: "text/javascript" "var x =
5;"))
<script type="text/javascript">
//<![CDATA[
var x = 5;
//]]>
</script>
```

```
(style/inline v ...) → procedure?
  v : outputable/c
```

Produces a value that renders as an inline style sheet.

Example:

```
> (output-xml (style/inline type: "text/css"
                            ".racket { font-size: xx-large; }"))
<style type="text/css">
.racket { font-size: xx-large; }
</style>
```

2.1.1 Other HTML elements

```
(require scribble/html/extra)
```
package: scribble-html-lib

Provides renderers for HTML elements that are not provided by `scribble/html/html`.

```
(article v ...) → procedure?
  v : outputable/c
(aside v ...) → procedure?
  v : outputable/c
(audio v ...) → procedure?
  v : outputable/c
(bdi v ...) → procedure?
  v : outputable/c
(canvas v ...) → procedure?
  v : outputable/c
(data v ...) → procedure?
  v : outputable/c
(datalist v ...) → procedure?
  v : outputable/c
(figcaption v ...) → procedure?
  v : outputable/c
(figure v ...) → procedure?
  v : outputable/c
(footer v ...) → procedure?
  v : outputable/c
(header v ...) → procedure?
  v : outputable/c
(main v ...) → procedure?
  v : outputable/c
(map v ...) → procedure?
  v : outputable/c
```

```
(mark v ...) → procedure?
  v : outputable/c
(math v ...) → procedure?
  v : outputable/c
(meter v ...) → procedure?
  v : outputable/c
(nav v ...) → procedure?
  v : outputable/c
(output v ...) → procedure?
  v : outputable/c
(picture v ...) → procedure?
  v : outputable/c
(progress v ...) → procedure?
  v : outputable/c
(rb v ...) → procedure?
  v : outputable/c
(rp v ...) → procedure?
  v : outputable/c
(rt v ...) → procedure?
  v : outputable/c
(rtc v ...) → procedure?
  v : outputable/c
(ruby v ...) → procedure?
  v : outputable/c
(section v ...) → procedure?
  v : outputable/c
(summary v ...) → procedure?
  v : outputable/c
(svg v ...) → procedure?
  v : outputable/c
(template v ...) → procedure?
  v : outputable/c
(time v ...) → procedure?
  v : outputable/c
(video v ...) → procedure?
  v : outputable/c
```

Like `element/not-empty`, but with the symbolic form of the function name added as the first argument.

Example:

```
> (output-xml (title "The Book"))
<title>The Book</title>
```

```
(source v ...) → procedure?
  v : outputable/c
(track v ...) → procedure?
  v : outputable/c
```

Like `element`, but with the symbolic form of the function name added as the first argument.

Example:

```
> (output-xml (hr))
<hr />
```

2.2 Generating XML Strings

```
(require scribble/html/xml)
```
package: `scribble-html-lib`

The `scribble/html/xml` provides functions for XML representations that *XML-render* to string form via `output-xml` or `xml->string`.

```
(output-xml content [port]) → void?
  content : outputable/c
  port : output-port? = (current-output-port)
```

Renders *content* in the same way as `output`, but using the value of `xml-writer` as the current writer so that special characters are escaped as needed.

```
(xml->string content) → string?
  content : outputable/c
```

Renders *content* to a string via `output-xml`.

```
(xml-writer) → ((string? output-port? . -> . void))
(xml-writer writer) → void?
  writer : ((string? output-port? . -> . void))
```

A parameter for a function that is used with `with-writer` by `output-xml`. The default value is a function that escapes `&`, `<`, `>`, and `"` to entity form.

```
(make-element tag attrs content)
 → (and/c procedure outputable/c?)
  tag : symbol?
  attrs : (listof (cons/c symbol? outputable/c))
  content : outputable/c
```

Produces a value that XML-renders as XML for the given tag, attributes, and content.

When an attribute in *attrs* is mapped to `#f`, then it is skipped. When an attribute is mapped to `#t`, then it is rendered as present, but without a value.

Examples:

```
> (output-xml (make-element 'b '() '("Try" #\space "Racket")))
<b>Try Racket</b>
> (output-xml (make-element 'a '((href . "http://racket-
lang.org")) "Racket"))
<a href="http://racket-lang.org">Racket</a>
> (output-xml (make-element 'div '((class . "big") (overlay .
#t)) "example"))
<div class="big" overlay>example</div>
```

```
(element tag attrs-and-content ...)
 → (and procedure outputable/c?)
  tag : symbol?
  attrs-and-content : any/c
```

Like `make-element`, but the list of *attrs-and-content* is parsed via `attributes+body` to separate the attributes and content.

Examples:

```
> (output-xml (element 'b "Try" #\space "Racket"))
<b>Try Racket</b>
> (output-xml (element 'a 'href: "http://racket-
lang.org" "Racket"))
<a href="http://racket-lang.org">Racket</a>
> (output-xml (element 'div 'class: "big" 'overlay: #t "example"))
<div class="big" overlay>example</div>
> (require scribble/html)
> (output-xml (element 'div class: "big" overlay: #t "example"))
<div class="big" overlay>example</div>
```

```
(element/not-empty tag
                   attrs-and-content ...)
 → (and/c procedure? outputable/c)
  tag : symbol?
  attrs-and-content : any/c
```

Like `element`, but the result always renders with an separate closing tag.

Examples:

```
> (output-xml (element 'span))
<span />
> (output-xml (element/not-empty 'span))
<span></span>
```

```
(attribute? v) → (or/c #f symbol?)
  v : any/c
```

Returns a symbol without if *v* is a symbol that ends with `:`, `#f` otherwise. When a symbol is returned, it is the same as *v*, but without the trailing `:`.

Examples:

```
> (attribute? 'a:)
'a
> (attribute? 'a)
#f
> (require scribble/html)
> (attribute? a:)
'a
```

```
(attributes+body lst) → (listof (cons/c symbol? any/c)) list?
  lst : list?
```

Parses *lst* into an association list mapping attributes to list elements plus a list of remaining elements. The first even-positioned (counting from 0) non-`attribute?` element of *lst* is the start of the "remaining elements" list, while each preceding even-positioned attribute is mapped in the association list to the immediately following element of *lst*. In the association list, the trailing `:` is stripped for each attribute.

```
(split-attributes+body lst) → list? list?
  lst : list?
```

Like `attributes+body`, but produces a flat list (of alternating attributes and value) instead of an association list as the first result.

```
(literal content ...) → procedure?
  content : any/c
```

Produces a value that XML-renders without escapes for special characters.

Examples:

```
> (output-xml (literal "a->b"))
a->b
> (output-xml "a->b")
a-&gt;b
```

```
(entity v) → procedure?
  v : (or/c exact-integer? symbol?)
```

Produces a value that XML-renders as a numeric or symbolic entity.

Example:

```
> (output-xml (entity 'gt))
&gt;
```

```
(comment content ... [#:newlines? newlines?]) → procedure?
  content : outputable/c
  newlines? : any/c = #f
```

Produces a value that XML-renders as a comment with literal content. If *newlines?* is true, then newlines are inserted before and after the content.

Example:

```
> (output-xml (comment "testing" 1 2 3))
<!--testing123-->
```

```
(cdata  content
        ...
       [#:newlines? newlines?
        #:line-pfx line-pfx]) → procedure?
  content : outputable/c
  newlines? : any/c = #t
  line-pfx : any/c = #f
```

Produces a value that XML-renders as CDATA with literal content. If *newlines?* is true, then newlines are inserted before and after the content. The *line-pfx* value is rendered before the CDATA opening and closing markers.

Example:

```
> (output-xml (cdata "testing" 1 2 3))
<![CDATA[
testing123
]]>
```

```
(define/provide-elements/empty tag-id ...)
```

Defines and exports *tag-id* as a function that is like `element`, but with `'tag-id` added as the first argument.

```
(define/provide-elements/not-empty tag-id ...)
```

Defines and exports *tag-id* as a function that is like `element/not-empty`, but with `'tag-id` added as the first argument.

```
(define/provide-entities entity-id ...)
```

Defines and exports *entity-id* as the result of `(entity 'entity-id)`.

2.3 HTML Resources

```
(require scribble/html/resource)
          package: scribble-html-lib
```

```
(resource path renderer [#:exists exists])
 → (and/c resource?
          (->* () (outputable/c) -> string?))
  path : string?
  renderer : (or/c (path-string? . -> . any) #f)
  exists : (or/c 'delete-file #f) = 'delete-file
```

Creates and returns a new *resource* value. Creating a resource registers *renderer* (if non-`#f`) to be called when rendering is initiated by `render-all`, while calling the result resource as a function generates a URL for the resource.

For example, a typical use of `resource` is to register the generation of a CSS file, where the value produced by `resource` itself renders as the URL for the generated CSS file. Another possible use of `resource` is to generate an HTML file, where the `resource` result renders as the URL of the generated HTML page.

The *path* argument specifies the path of the output file, relative to the working directory, indicating where the resource file should be placed. Though `url-roots`, *path* also determines the ultimate URL. The *path* string must be a `/`-separated relative path with no `..`, `.`, or `//`. The *path* string can end in `/`, in which case `"index.html"` is effectively added to the string. Using `resource` with `#f` as *renderer* is useful for converting a path to a URL according to `url-roots`.

The *renderer* argument (when non-`#f`) renders the resource, receiving the path for the file to be created. The path provided to *renderer* will be different from *path*, because the function is invoked in the target directory.

The resulting resource value is a function that returns the URL for the resource. The function accepts an optional boolean; if a true value is provided, the result is an absolute URL, instead of relative. Note that the function can be used as a value for `output`, which uses the resource value as a thunk (that renders as the relative URL for the resource). The default relative resulting URL is, of course, a value that depends on the currently rendered resource that uses this value.

When *renderer* is called by `render-all`, more resources can be created while rendering; the newly created resources will also be rendered, in turn, until no more new resources are created.

If *exists* is `'delete-file` and the target file exists when *renderer* is to be called, then the file is deleted before *renderer* is called.

```
(url-roots)
 → (or/c #f
         (listof (cons/c path-string?
                         (cons/c string?
                                 (listof (or/c 'abs 'index))))))
(url-roots roots) → void?
  roots : (or/c #f
                (listof (cons/c path-string?
                                (cons/c string?
                                        (listof (or/c 'abs 'index))))))
```

A parameter that determines how resource paths are converted to URLs for reference. A `#f` value is equivalent to an empty list.

The parameter value is a mapping from path prefixes to URLs (actually, any string). When two paths have the same prefix, links from one to the other are relative (unless absolute links are requested); if they have different prefixes, the full URL is used. The paths enclosed by two root paths must be disjoint (e.g., the list must not include both `"/colors"` and `"/colors/red"`, but it can include both `"/colors/red"` and `"/colors/blue"`).

If an item in the parameter's list includes `'abs`, then a site-local, absolute URL (i.e., a URL that starts with `/`) is produced for references among files within the corresponding prefix.

If an item in the parameter's list includes `'index`, then a reference to a directory path is converted to a reference to `"index.html"`, otherwise a reference to `"index.html"` is converted to a directory path.

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

Returns `#t` if *v* is a procedure (that takes 0 or 1 arguments) produced by `resource`.

```
(render-all) → void?
```

Generates all resources registered via `resource`.

```
(file-writer content-writer content) → (path-string? . -> . any)
  content-writer : (outputable/c output-port? . -> . any)
  content : outputable/c
```

Produces a function that is useful as a *writer* argument to `resource`. Given a path, the produced function writes *content* to the path by passing *content* and an output port for the file to *content-writer*.

Index