Skribilo
User Manual

for version 0.9.3

Érick Gallesio, Manuel Serrano, Ludovic Courtès
http://nongnu.org/skribilo/

Contents

Introduction

Skribilo is a document production toolkit and a programming language designed for implementing electronic documents[1]. It is mainly designed for the writing of technical documents such as the documentation of computer programs. With Skribilo these documents can be rendered using various tools and technologies. For instance, a Skribilo document can be *compiled* to an HTML file that suits Web browser, it can be compiled to a TeX file in order to produce a high-quality printed document, and so on.

This manual documents Skribilo version 0.9.3. Since it is based on Skribe's user manual, you might stumble upon documentation bits that are obsolete or inaccurate in the context of Skribilo, although work is being done to fix it.

Who May Use Skribilo?

Anyone needing to design web pages, PostScript/PDF files or Info documents can use Skribilo. In particular, there is *no need* for programming skills in order to use Skribilo. Skribilo can be used as any text description languages such as LaTeX, Lout or HTML.

Why Use Skribilo?

There are three main reasons for using Skribilo:

- It is easier to type in Skribilo texts than other text description formats. The need for *meta keyword*, that is, words used to describe the structure of the text and not the text itself, is very limited.

- Skribilo is highly skilled for computing texts. It is very common that one needs to automatically produce parts of the text. This can be very simple such as, for instance, the need to include inside a text, the date of the last update or the number of the last revision. Sometimes it may be more complex. For instance, one may be willing to embed inside a text the result of a complex arithmetic computation. Or even, you may want to include some statistics about that text, such as, the number of words, paragraphs, sections, and so on. Skribilo makes these sort of text manipulation easy whereas other systems rely on the use of text preprocessors.

- The same source file can be compiled to various output formats such as HTML, PostScript, Info pages, etc.

[1] To be more precise, the programming language itself is that of Skribe, the project Skribilo is based on.

More on Skribilo

Skribilo is based on Skribe, which was designed and implemented by Manuel Serrano and Érick Gallesio. Although it departs from it on some aspects, it shares the overall design and philosophy. Érick and Manuel described the main design decisions behind Skribe in a paper published in the 2005 Journal of Functional Programming (JFP) entitled *Skribe: A Functional Authoring Language*. Although parts of the paper are slightly outdated, it gives a very good idea of Skribilo's innards, and notably contains a description of the 3 stages of documentation "evaluation".

Chapter 1. Getting Started

In this chapter, the syntax of a Skribilo text is presented *informally*. In particular, the Skribilo syntax is compared to the HTML syntax. Then, it is presented how one can use Skribilo to make dynamic text (i.e texts which are generated by the system rather than entered-in by hand). Finally, It is also presented how Skribilo source files can be processed.

1.1. Hello World!

In this section we show how to produce very simple electronic documents with Skribilo. Suppose that we want to produce the following Web document:

> ### Hello World!
>
> This is a very simple text.

The HTML source file for such a page should look like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Hello World Example</TITLE>
</HEAD>
<BODY>
<H1>Hello World!</H1>

This is a very simple text.
</BODY>
</HTML>
```

In Skribilo, the very same document must be written:

```
(document :title [Hello World!]
   :html-title [Hello World Example]

   [This is a very simple text.])
```

1.2. Adding Colors and Fonts

Let us suppose that we want now to colorize and change the face of some words such as:

Hello World!

This is a **very** *simple* text.

The HTML source file for such a document should look like:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN">
<HTML>
<HEAD>
<TITLE>Hello World Example</TITLE>
</HEAD>
<BODY>
<H1>Hello World!</H1>

This is a <B>very</B> <I>simple</I> <FONT color="red">text</FONT>.
</BODY>
</HTML>
```

In Skribilo, the very same document must be written:

```
(document :title [Hello World!]
   :html-title [Hello World Example]

   [This is a ,(bold [very]) ,(it [simple])
,(color :fg [red] [text]).])
```

As one may notice the Skribilo version is much more compact than the HTML one.

1.3. Structured Documents

For large documents there is an obvious need of structure. Skribilo documents may contain **chapters**, **sections**, **subsections**, **itemize**, … For instance, if we want to extend our previous example to:

Hello World!

1. A first Section

This is a **very** *simple* text.

2. A second Section

That contains an **itemize** construction:
. first item
. second item
. third item

The Skribilo source for that text is:

```
(document :title [Hello World!]

   (chapter :title [A first Section]

      [This is a ,(bold [very]) ,(it [simple])
,(color :fg [red] [text]).])

   (chapter :title [A second Section]

      [That section contains an ,(bold [itemize])
construction:

 ,(itemize (item [first item])
           (item [second item])
           (item [third item])))])
```

1.4. Hyperlinks

A Skribilo document may contain links to chapters, to sections, to other Skribilo documents or web pages. The following Skribilo source code illustrates these various kinds of links:

```
(document :title [Various Links]

  (chapter :title [A Section]

     [The first link points to an external web page.
Here we point to a ,(ref :url "http://slashdot.org/"
:text [Slashdot]) web page.  The second one points to
the second ,(ref :chapter "A Second Section" :text
[section]) of that document.])
```

```
  (chapter :title "A Second Section"

     [The last link points to the first
,(ref :skribe "user.sui" :chapter "Introduction"
:text [chapter]) of the Skribilo User Manual.]))
```

1.5. Using Modules

Skribilo comes with extra features bundled in *modules*. In fact, anyone can write new modules that extend Skribilo. For example, extra bibliography features are bundled in a module called `(skribilo biblio)`. To use them in a document, that document the following line must be added at the very beginning of the document, before the `document` markup:

```
(use-modules (skribilo biblio))
```

Suppose you also want to use the mathematical formula layout package, which is bundled in the `(skribilo package eq)` module. To do that, you can either add another `use-modules` form at the top of the document, or combine both:

```
(use-modules (skribilo biblio)
             (skribilo package eq))
```

The module system described above is actually that of GNU Guile. More information is available in Guile's manual.

1.6. Dynamic Documents

Since Skribilo is a programming language, rather than just a markup language, it is easy to use it to generate some parts of a document. This section presents here the kind of documents that can be created with Skribilo.

1.6.1. Simple Computations

In this section we present how to introduce a simple computation into a document. For instance, the following sentence

Document creation date: Wed Jan 13 09:14:49Z 2016

is generated with the following piece of code (using SRFI-19):

```
[Document creation date: ,(date->string (current-date))]
```

Here, we use the Skribilo function `date` to compute the date to be inserted in the document. In general, any valid Scheme expression is authorized inside a `,(...)` construct[1].

Another example of such a computation is given below.

```
[The value of ,(symbol "pi") is ,(* 4 (atan 1))]
```

When evaluated, this form produces the following output:

The value of π is 3.141592653589793.

1.6.2. Text Generation

When building a document, one often need to generate some repetitive text. Skribilo programming skills can be used to ease the construction of such documents as illustrated below.

- The square of **1** is **1**
- The square of **2** is **4**
- The square of **3** is **9**
- The square of **4** is **16**
- The square of **5** is **25**
- The square of **6** is **36**
- The square of **7** is **49**
- The square of **8** is **64**
- The square of **9** is **81**

This text has been generated with the following piece of code

```
(itemize
 (map (lambda (x)
        (item [The square of ,(bold x) is ,(bold (* x x))]))
      '(1 2 3 4 5 6 7 8 9)))
```

[1] Any valid GNU Guile Scheme code may be used inside `,(...)` expressions!

1.6.3. Introspection

In Skribilo, a document is represented by a tree which is available to the user. So, it is easy to perform introspective tasks on the current document. For instance the following code displays as an enumeration the sections titles of the current chapter:

```
(resolve (lambda (n e env)
             (let* ((current-chapter (ast-chapter n))
                    (body  (markup-body current-chapter))
                    (sects (filter (lambda (x) (is-markup? x 'sec-
tion))
                                   body)))
                (itemize
                 (map (lambda (x)
                         (item (it (markup-option x :title))))
                      sects)))))
```

Without entering too much into the details here, the resolve function is called at the end of the document processing. This function searches the node representing the chapter to which belongs the current node and from it finds all its sections. The titles of these sections are put in italics in an itemize.

The execution of this code yield the following text:

- *Hello World!*
- *Adding Colors and Fonts*
- *Structured Documents*
- *Hyperlinks*
- *Using Modules*
- *Dynamic Documents*
- *Compiling Skribilo Documents*

1.7. Compiling Skribilo Documents

There are several ways to render a Skribilo document. It can be statically compiled by the `skribilo` compiler to various formats such as HTML, LaTeX, Lout and so on. In this section we only present static "document compilation".

Let us suppose a Skribilo text located in a file `file.skb`. In order to compile to various formats one must type in:

```
$ skribilo –target=html file.skb -o file.html # This produces an HTML file.
$ skribilo -t latex file.skb -o file.tex  # This produces a TeX file.
$ skribilo -t lout file.skb -o file.lout # This produces a Lout file.
```

The reference of the `skribilo` compiler is given in Chapter 14.

Chapter 2. Syntax

This chapter describes the syntax or Skribilo documents—or rather, the available syntaxes Skribilo documents can use. Skribilo actually supports several input syntaxes, each of which is implemented by a *reader*. The input syntax (and reader) can be selected at document compilation time using the `-reader` option of the compiler (see Chapter 14). Nevertheless, it has a "preferred" syntax (the default syntax), which is that of the Skribe document preparation system. Thus, the Skribe syntax is first described, and then alternate syntaxes are presented.

2.1. The Skribe Syntax

By default or when the `-reader=skribe` option is passed to the compiler, a Skribilo document is composed of *Skribe expressions*, which resemble expressions in the Scheme programming language, with a few extensions to make them more convenient to use within documents. A Skribe expression can be:

- An atomic expression, such as a string of characters, a number.
- A list.
- A text.

Here are several examples of correct Skribe expressions:

- `"foo"`, a string of characters composed of the characters 'f', 'o' and 'o'.
- `123 3.14`, two numbers.
- `#t #f`, the *true* and *false* Skribe value.
- `(bold "foo bar")`, a list.
- `[A text sample]`, a simple text containing three words and no escape sequence.
- `[Another text sample (that is still) simple]`, another simple text.
- `[Another ,(bold "text") sample]`, a more complex text that contains two words (`Another` and `sample`) and an expression `(bold "text")`. The escape sequence is introduced with the '`,(`' characters.

Expressions are evaluated, thus `(bold "foo")` has the effect of typesetting the word `foo` in bold face to produce **foo**. Escape sequences enable evaluation of expressions inside the text. Thus the text `[Another ,(bold "text") sample]` produces '`Another` **`text`** `sample`'. On the other hand `[Another (bold "text") sample]` produces '`Another (bold "text") sample`' because it does not contain the escape sequence ',('.

2.1.1. Formal Syntax

<expr> –> <atom>
| <text>
| <list>
<list> –> (<expr>+)
<text> –> [*any sequence but ',(' or a ',<list>'*]
<atom> –> <boolean>
| <integer>
| <float>
| <string>
| <color>
<integer> –> `[0-9]+`
<float> –> `[0-9]+.[0-9]*`
| `[0-9]*.[0-9]+`
<string> –> …
<color> –> <string>
| `#[0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f][0-9a-f]`

2.1.2. Values

2.1.2.1. Width

A Skribe *width* refers to the horizontal size a construction occupies on an output document. There are three different ways for specifying a width:

An absolute pixel size
: This is represented by an *exact* integer value (such as `350`).

A relative size
: This is represented by an *inexact* integer value (such as `50.0`) which ranges in the interval [-100.0 .. 100.0]

An engine dependent representation
: This is represented by a string that is directly emitted in the output document (such as HTML column `"0*"` specification). Note that this way of specifying width is strictly unportable.

2.2. The Outline Syntax

Alternatively, Skribilo allows documents to be written in a plain text format, with almost no markup. Instead, conventions borrowed from Emacs' Outline Mode to denote the text structure as well as other common conventions are used to express various formatting ideas. This syntax is implemented by the `outline` reader; thus, it is made available by passing the `-reader=outline` option to the compiler. The major elements of this syntax are the following:

Document title and author
: The document title is introduced by adding a line starting with `Title:` at the beginning of the text file, and followed by the title. Likewise, the author can be specified with a line starting with `Author:`.

Sectioning
: Chapters are introduced using a heading preceding by a single `*` (star) character. For instance, `* The First Part` on a line on its own, followed by an empty line, introduces a new chapter entitled "The First Part". Likewise, two stars introduce a section, three stars introduce a subsection, etc.

Emphasis, italics, bold
: Words or phrases surrounded by the _ (underscore) character are emphasized; those surrounded by `/` (slash) characters are italicized; finally, those surrounded by `*` (star) characters are typeset in boldface (see Section 3.5).

Quotes
: Words enclosed in double quotes (i.e., two back-quote characters, then two single-quote characters) are interpreted as quoted text, as per `q`.

Code
: Words enclosed in single quotes (i.e., one back-quote character, then one single-quote) are interpreted as code and are typeset using a fixed-width font, as per `tt`.

Hyperlinks
: URLs are automatically recognized and converted into a `(ref :url …)` form (see `ref`). In addition, `outline` has limited support for Org-Mode-style hyperlinks; for instance, `[[http://gnu.org/][The GNU Project]]` yields The GNU Project.

Here is an example showing how the `outline` syntax maps to the native `skribe` syntax:

Example 1. The `outline` syntax

```
-*- mode: outline; coding: iso-8859-1; -*-

Title: Demonstrating Skribilo's Outline Syntax
```

```
Author: Ludovic Courtes
Keywords: Skribilo outline Emacs

This document aims to *demonstrate*
[[http://skribilo.nongnu.org/][Skribilo]]'s outline syntax.


* The First Chapter

The first chapter contains a couple of sections.
They look as though they had been introduced with
the 'section' markup of the Skribe syntax.

** The First Section

This section is pretty much empty.

** The Second Section

This section introduces lists.

*** Bullet Lists

This section contains a wonderful 'itemize'-style bullet list:

  - the first item;

  - the second item;

  - the last one, which spans
    two lines of text.


And that's it.  Note that list items had to be
separated by empty lines.

*** Enumerations

This section shows an 'enumerate'-style list:

  1. The first item;

  2. The second one;

  3. The last one.


Note that list items are numbered this time.


* The Second Chapter

The second chapter does _not_ add anything useful.

Text like this that starts after an empty line is
put into a new paragraph.
```

…produces:

```
(document
  #:title
  "Demonstrating Skribilo's Outline Syntax"
  #:author
```

```
  (author #:name "Ludovic Courtes")
  #:keywords
  '("Skribilo outline Emacs")
  (p (list (list (list "This document aims to "
                       (bold "demonstrate")
                       "")
                 "\n")
           (list ""
                 (ref #:url
                      "http://skribilo.nongnu.org/"
                      #:text
                      "Skribilo")
                 "'s outline syntax.")
           "\n"))
  (chapter
    #:title
    "The First Chapter"
    (p (list (list (list "The first chapter contains a couple of
sections."
                         "\n")
                   "They look as though they had been introduced
with"
                   "\n")
             (list "the "
                   (tt "section")
                   " markup of the Skribe syntax.")
             "\n"))
    (section
      #:title
      "The First Section"
      (p (list "This section is pretty much empty." "\n")))
    (section
      #:title
      "The Second Section"
      (p (list "This section introduces lists." "\n"))
      (subsection
        #:title
        "Bullet Lists"
        (p (list (list "This section contains a wonderful "
                       (tt "itemize")
                       "-style bullet list:")
                 "\n"))
        (itemize
          (item (list "the first item;"))
          (item (list "the second item;"))
          (item (list (list "the last one, which spans")
                      "    two lines of text."))))
```

```
          (p (list (list "And that's it.  Note that list items had
to be"
                         "\n")
                   "separated by empty lines."
                   "\n")))
        (subsection
          #:title
          "Enumerations"
          (p (list (list "This section shows an "
                         (tt "enumerate")
                         "-style list:")
                   "\n"))
          (enumerate
            (item (list "The first item;"))
            (item (list "The second one;"))
            (item (list "The last one.")))
          (p (list "Note that list items are numbered this time."
                   "\n")))))
    (chapter
      #:title
      "The Second Chapter"
      (p (list (list "The second chapter does "
                     (emph "not")
                     " add anything useful.")
               "\n"))
      (p (list (list "Text like this that starts after an empty line
is"
                     "\n")
               "put into a new paragraph."
               "\n"))))
```

The `outline` mode makes it possible to quickly create documents that can be output in variety of formats (see Chapter 13). The downside is that, being a very simple markup-less document format, there are many things that cannot be done using it, most notably tables, bibliographies, and cross-references.

2.3. The RSS 2.0 Syntax

RSS 2.0 (aka. *Really Simple Syndication*) is supported as an input syntax. To use it, just pass `-reader=rss-2` to the compiler. This makes it possible to generate Skribilo documents from RSS 2.0 feeds, which can be useful or at least funny. Consider the following example:

```
$ wget http://planet.gnu.org/rss20.xml
$ skribilo -R rss-2 -t lout -c column-number=2 < rss20.xml \
```

```
|lout |ps2pdf - > gnu-planet.pdf
```

It produces a two-column PDF file with the contents of the RSS feed of GNU Planet, where each item of the feed is mapped to a Skribilo "chapter".

2.4. Documents in Scheme Programs

It is also possible to create and output Skribilo documents from a Guile Scheme program. In that case, you get to use the Scheme syntax, which is close to the Skribe syntax described above, modulo the `[…]` constructs. A typical Scheme program that would produce and output a document, pretty much like what the `skribilo` compiler does, would look like this:

Example 2. Programming Skribilo documents in Scheme.
This should give you an idea of the wonderful, life-changing things that can be achieved with a bit of *document programming*.

```
(use-modules (skribilo engine)       ;; provides 'find-engine'
             (skribilo evaluator)    ;; provides 'evaluate-document'
             (skribilo package base) ;; provides 'chapter', etc.
             (srfi srfi-1))

(let (;; Select an engine, i.e., an output format.
      (e (find-engine 'html))

      ;; Create a document.
      (d (document #:title "Some Title"
            (chapter #:title "The Chapter"
               (p "The paragraph…  "
                  "Text consists of "
                  "a list of strings.")
               (apply itemize
                      (map number->string
                           (iota 10)))))))

  ;; "Evaluate" the document to an HTML file.
  (with-output-to-file "foo.html"
    (lambda ()
      (evaluate-document d e))))
```

2.5. Writing New Readers

Skribilo is extensible and makes it easy to add *custom readers*, allowing the use of virtually any input syntax. A reader is essentially a procedure like R5RS' `read`, i.e., a one-argument procedure that takes a port and returns an S-expression. The returned S-expression should be a valid "document program" as shown in 2.4.

The are a few additional details, though. Implementations of readers are required to use the `(skribilo reader)` modules and the `define-reader` macro. In addition, the reader must live in its own module, under the `(skribilo reader)` module hierarchy, so that the

reader lookup mechanism (used by the `-reader` option of the compiler) can find it. This mechanism is the same as that used for engines (see Section 13.1.5). A skeleton for a reader would be like this:

Example 3. Writing a new reader.
Users and encouraged to look at examples in the Skribilo source for additional details.

```
(define-module (skribilo reader my-reader)
  :use-module (skribilo reader)
  :export (reader-specification))

(define (make-my-reader)
  (lambda (port)
    …))

(define-reader my-reader  ;; the reader name
  "0.1"                   ;; a version number
  make-my-reader)         ;; the procedure that returns
                          ;; a reader procedure
```

Chapter 3. Standard Markups

This chapter describes the forms composing Skribilo texts that use the Skribe syntax (see Section 2.1). In XML/HTML jargon these forms are called *markups*. In LaTeX they are called *macros*. In Skribilo these forms are called *functions*. In this manual, we will say that we *call a function* when a function is used in a form. The values used in a function call are named the *actual parameters* of the function or *parameters* in short. When calling a function with parameters we say that we are *passing* arguments to the function.

In this document function names are typeset in boldface. We call *keyword argument* a named argument, i.e., an argument whose name, starting with a colon (`:`), must be specified when the function is called. Other arguments are called *plain arguments* or *arguments* for short. An *optional argument* is represented by a list, starting with the character « [» and ending with the character «] », whose first element is a keyword argument and the optional second (`#f` when not specified) element is the default value used if the optional argument value is not provided on a function call. Arguments that are not optional are said *mandatory*. If a plain argument is preceded by a `.` character, this argument may be used to accumulate several values. There are two ways to pass actual arguments to a function:

- for keyword arguments: the value of the parameter must be preceeded by the name of the argument.
- for plain arguments: a value is provided.

Example: Let us consider the function `section` defined as follows:

```
(section :title [:number #t] [:toc #t] . body)
```

The argument **:title** is a mandatory keyword argument. The keyword arguments **:number** and **:toc** are optional. The plain argument `body` is preceeded with a `.` character so it may receive several values. All the following calls are legal `section` calls:

```
(section :title "A title" "This is the body of the section")
(section :title "A title" "This" " is" " the body of the section")
(section :title "A title" :number 3 "This" " is" " the body of the
section")
(section :title "A title" :toc #f :number 3 "This" " is" " the body of
the section")
(section :title "A title" :number 3 :toc #f "This" " is" " the body of
the section")
```

The remainder of this chapter describes "standard" markups or functions that are commonly used in documents. By "standard", we mean two things: first, you will quickly notice that they look familiar if you have ever written, say, HTML or LaTeX documents; second, they are standard because these markups are always available by default to Skribilo documents, unlike those bundled in separate packages such as pie charts, slides, etc. In fact, these markups are also bundled in a package, called `base`, but this package is always available to Skribilo documents[1].

3.1. Building documents

3.1.1. Document

The `document` function defines a Skribe document.

```
(document [:env '()] [:keywords '()] [:ending] [:author] [:html-title]
    [:title] [:class "document"] [:ident] node…)
```

`:ident` — html lout latex context info xml

The node identifier.

`:class` — html lout latex context info xml

The node class.

`:title` — html lout latex context info

The title of the document.

`:html-title` — html

The title of window of the HTML browser.

`:author` — html lout latex context info

The authors of the document.

`:ending` — html lout latex context info

An ending text.

`:keywords` — html lout

A list of keywords which may be plain strings or markups. The keywords will not appear in the final document but only as meta-information (e.g., using the HTML 'meta' tag) if the engine used supports it.

`:env` — html lout latex context

A counter environment.

`node…`

The document nodes.

[1] When creating Skribilo documents within Guile Scheme programs (see Section 2.4), these standard markups can be made available by using the following clause: `(use-modules (skribilo package base))`.

See also author, p. 21 , chapter, p. 23 , toc, p. 26 .

Example 4. The document markup

```
(document :title "This is a Skribilo document"
          :author (list (author :name "Foo" :email (mailto
"foo@nowhere.org"))
                        (author :name "Bar" :email (mailto
"bar@anywhere.org"))
                        (author :name "Gee" :email (mailto
"gee@nowhere.org")))
  "A body…")
```

… produces:

This is a Skribilo document

Foo <*foo@nowhere.org*> Bar <*bar@anywhere.org*> Gee <*gee@nowhere.org*>
A body…

3.1.2. Author

The author function is used to specify the authors of a Skribe document.

```
(author :name [:align 'center] [:photo] [:phone] [:address] [:url] [:email]
    [:affiliation] [:title] [:class "author"] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:name` html lout latex context info

The name of the author.

`:title` html lout latex context info

His title.

`:affiliation` html lout latex context info

His affiliation.

`:email` html lout latex context info

His email.

`:url` html lout latex context info

His web page.

`:address` html lout latex context info

His address.

`:phone` html lout latex context info

His phone number.

`:photo` html lout latex context info

His photograph.

`:align` html lout latex context info

The author item alignment.

See also `mailto`, p. 59 , `ref`, p. 56 .

Example 5. The author markup

```
(author :name "Manuel Serrano"
        :affiliation "Inria Sophia-Antipolis"
        :url (ref :url "http://www.inria.fr/mimosa/Manuel.Serrano")
        :email (mailto "Manuel.Serrano@inria.fr")
        :address `("2004 route des Lucioles - BP 93"
                   "F-06902 Sophia Antipolis, Cedex"
                   "France")
        :phone "phone: (+33) 4 92 38 76 41")
```

… produces:

Manuel Serrano <*Manuel.Serrano@inria.fr*>
Inria Sophia-Antipolis
2004 route des Lucioles - BP 93F-06902 Sophia Antipolis, CedexFrance
phone: (+33) 4 92 38 76 41
http://www.inria.fr/mimosa/Manuel.Serrano

3.2. Spacing

By default, the spacing rules are the same as those of TeX/LaTeX, for instance: subsequent white spaces in the input text are coalesced into a single space in the output. Language-dependent spacing rules (e.g., for punctuation) are to be handled by the engine. This frees you from having to remember all the spacing details.

Additionally, the following markups allow you to produce explicit unbreakable and breakable space, respectively:

```
(~ [:class])
```

`:class` html lout latex context info xml

The node class.

```
(breakable-space [:class] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

3.3. Sectioning

3.3.1. Chapter

The function `chapter` creates new chapters.

```
(chapter :title [:number #t] [:toc #t] [:file] [:info-node] [:html-title]
    [:class "chapter"] [:ident] node…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:title` html lout latex context info

The title of the chapter.

`:html-title` html

The title of window of the HTML browser.

`:info-node` info

The title of the Info node (see Section about the Info engine).

`:number` html lout latex context info

This argument controls the chapter number. A value of #t means that Skribilo computes the chapter number. A value of #f means that the chapter has no number. A number or string specifies a number to be used in lieu of the automatically computed number.

`:toc` html lout latex context info

This argument controls if the chapter must be referenced in the table of contents.

`:file` html lout latex context info

The argument must be a boolean or a string. A value of #t tells the compiler to store that chapter in a separate file; a value of #f tells the compiler to embed the chapter in the main target file. When the argument is a string, it is used as the name of the file for this chapter.

`node...`

The nodes of the chapter.

See also `document`, p. 20 , `section`, p. 24 , `toc`, p. 26 .

Example 6. The `chapter` markup

```
(chapter :title "This is a simple chapter" :number #f :toc #f
   (p [Its body is just one sentence.]))
```

... produces:

This is a simple chapter

Its body is just one sentence.

3.3.2. Sections

These functions create new sections.

```
(section :info-node :title [:number #t] [:toc #t] [:file] [:class "section"]
    [:ident] node…)
(subsection :info-node :title [:number #t] [:toc #t] [:file] [:class
    "subsection"] [:ident] node…)
(subsubsection :info-node :title [:number #t] [:toc] [:file] [:class
    "subsubsection"] [:ident] node…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:title` html lout latex context info

The title of the chapter.

`:info-node` info

The title of the Info node (see Section about the Info engine).

`:number` html lout latex context info

This argument controls the chapter number. A value of #t means that Skribilo computes the chapter number. A value of #f means that the chapter has no number. A number or string specifies a number to be used in lieu of the automatically computed number.

`:toc` html lout latex context info

This argument controls if the chapter must be referenced in the table of contents.

`:file` html lout latex context info

The argument must be a boolean or a string. A value of #t tells the compiler to store that section in a separate file; a value of #f tells the compiler to embed the section in the main target file. When the argument is a string, it is used as the name of the file for this section.

`node…`

The nodes of the section.

See also `document`, p. 20 , `chapter`, p. 23 , `paragraph`, p. 25 , `toc`, p. 26 .

3.3.3. Paragraph

The function `paragraph` (also aliased `p`) creates paragraphs.

```
(paragraph [:class] [:ident] node…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`node…`

The nodes of the paragraph.

See also `document`, p. 20 , `chapter`, p. 23 , `section`, p. 24 , `p`, p. 26 .

The function `p` is an alias for `paragraph`.

(p `:ident` `[:class]` `node…`**)**

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`node…`

The nodes of the paragraph.

See also `document`, p. 20 , `chapter`, p. 23 , `section`, p. 24 , `paragraph`, p. 25 .

3.3.4. Blockquote

The function `blockquote` can be used for text quotations. A text quotation is generally renderd as an indented block of text.

(blockquote `[:class]` `[:ident]` `node…`**)**

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`node…`

The nodes of the quoted text.

3.4. Table of contents

The production of table of contains.

```
(toc [:subsubsection] [:subsection] [:section #t] [:chapter #t] [:class
    "toc"] [:ident] handle)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:chapter` html lout

A boolean. The value #t forces the inclusion of chapters in the table of contents.

`:section` html lout

A boolean controlling sections.

`:subsection` html lout

A boolean controlling subsections.

`:subsubsection` html

A boolean controlling subsubsections.

`handle`

An optional handle pointing to the node from which the table of contents if computed.

See also `document`, p. 20 , `chapter`, p. 23 , `section`, p. 24 , [?mark resolve: skribilo/documentation/api.scm:766:36:] , [?mark handle: skribilo/documentation/api.scm:766:36:] .

Example 7. The toc markup

```
(toc :chapter #t :section #f :subsection #f)
```

… produces:

The second example only displays the table of contents of the current chapter.

Example 8. A restricted table of contents

```
(resolve (lambda (n e env)
```

```
                (toc :chapter #t :section #t :subsection #t
                     (handle (ast-chapter n)))))
```

… produces:

3.5. Ornaments

Skribe supports the standard text ornaments.

(bold [**:class**] [**:ident**] node…**)**
(code [**:class**] [**:ident**] node…**)**
(emph [**:class**] [**:ident**] node…**)**
(it [**:class**] [**:ident**] node…**)**
(kbd [**:class**] [**:ident**] node…**)**
(roman [**:class**] [**:ident**] node…**)**
(sc [**:class**] [**:ident**] node…**)**
(sf [**:class**] [**:ident**] node…**)**
(sub [**:class**] [**:ident**] node…**)**
(sup [**:class**] [**:ident**] node…**)**
(tt [**:class**] [**:ident**] node…**)**
(underline [**:class**] [**:ident**] node…**)**
(var [**:class**] [**:ident**] node…**)**

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

node…

The nodes of the ornament.

Example 9. The ornament markups

```
```

```
(itemize (item (roman "a roman text."))
         (item (bold "a bold text."))
         (item (it "an italic text."))
         (item (emph "an emphasized text."))
         (item (underline "an underline text."))
         (item (kbd "a keyboard description."))
         (item (tt "a typewritter text."))
         (item (code "a text representing computer code."))
         (item (var "a computer program variable description."))
         (item (samp "a sample."))
         (item (sc "a smallcaps text."))
         (item (sf "a sans-serif text."))
         (item (sup "a superscripts text."))
         (item (sub "a subscripts text."))
         (item (underline (bold (it "an underline, bold, italic text.")))))
```

… produces:

- a roman text.
- **a bold text.**
- *an italic text.*
- *an emphasized text.*
- an underline text.
- a keyboard description.
- `a typewritter text.`
- `a text representing computer code.`
- `a computer program variable description.`
- a sample.
- A SMALLCAPS TEXT.
- a sans-serif text.
- a superscripts text.
- a subscripts text.
- ***an underline, bold, italic text.***

3.6. Line breaks

Line breaks and horizontal rules enable text cutting.

3.6.1. Linebreak

The Skribe function `linebreak` is unportable. Even if most engines handle it for their code production, using `linebreak` generally produces expected result. For instance, using `linebreak` with an engine producing LaTeX code is bound to fail. In consequence, as much as possible, one should prefer other ways for splitting a text

```
(linebreak [:class] [:ident] num)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`num`

The number of line breaks.

See also `paragraph`, p. 25 , `table`, p. 42 .

3.6.2. Horizontal rule

```
(hrule [:height 1] [:width 100.0] [:class] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:width` html context info

The 2.1.2.1 of the horizontal rule.

`:height` html context

The thickness of the rule. A positive integer value stands for a number of pixels.

3.7. Font

The function `font` enables font selection.

(font [:face] [:size] [:class] [:ident] node…**)**

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:size html lout latex context

The size of the font. The size may be relative (with respect to the current font size) or absolute. A relative font is either specified with a floating point value or a negative integer value. A positive integer value specifies an absolute font size.

:face html lout

The name of the font to be used.

node…

The nodes of the font.

Example 10. The font markup

```
(itemize
 (item (font :size -2 [A smaller font.]))
 (item (font :size 6 [An absolute font size.]))
 (item (font :size 4. [A larger font.]))
 (item (font :face "Helvetica" [An helvetica example.])))
```

… produces:

- A smaller font.
- An absolute font size.
- A larger font.
- An helvetica example.

3.8. Justification

These functions control the text layout. The default layout is text justification.

```
(flush :side [:class] [:ident] node…)
(center [:class] [:ident] node…)
(pre [:class] [:ident] node…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:side` html lout latex context info

The possible values for the side justification are left, center or right.

`node…`

The nodes of the font.

See also `linebreak`, p. 30 , `table`, p. 42 , `prog`, p. 75 .

Example 11. The justification markups

```
(center [A ,(linebreak) multilines ,(linebreak) text])
(hrule)
(flush :side 'left [A ,(linebreak) multilines ,(linebreak) text])
(hrule)
(flush :side 'right [A ,(linebreak) multilines ,(linebreak) text])
(hrule)
(pre [A text layout that

 preserves
 linebreaks          and spaces ,(it "(into which it is still legal")
,(it "to use Skribe markups)").
])
```

… produces:

A
multilines
text

A
multilines
text

A
multilines
text

A text layout that

preserves
linebreaks and spaces *(into which it is still legal to use Skribe markups).*

3.9. Enumeration

These functions implements three various style of enumerations.

```
(itemize :symbol [:class "itemize"] [:ident] item…)
(enumerate :symbol [:class "enumerate"] [:ident] item…)
(description :symbol [:class "description"] [:ident] item…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:symbol` html lout latex context info

The symbol that prefixes the items.

`item…`

The items of the enumeration.

Items are introduce by the means of the `item` markup:

```
(item :key [:class] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:key` html lout latex context info

The item key.

Example 12. The enumeration markups

```
(itemize (item [A first item.])
          (item [A ,(bold "second") one:
             ,(itemize (item "One.")
                       (item "Two.")
                       (item "Three."))])
          (item [Lists can be nested. For instance that item contains a
           ,(tt "description"):
           ,(description (item :key (bold "foo")
                               [is a usual Lisp dummy identifier.])
                         (item :key (bold "bar")
                               [is another one.])
                         (item :key (list (bold "foo") (bold "bar"))
                               [A description entry may contain more than
                                one keyword.]))])
          (item [The last ,(tt "itemize") entry contains an ,(tt
"enumerate"):
           ,(enumerate (item "One.") (item "Two.") (item "Three."))]))

(itemize :symbol "-"
          (item "One.")
          (item "Two.")
          (item "Three.")
          (item "Four."))
```

… produces:

- A first item.
- A **second** one:
  - One.
  - Two.
  - Three.
- Lists can be nested. For instance that item contains a `description`:

foo
is a usual Lisp dummy identifier.

bar
is another one.

foo bar
A description entry may contain more than one keyword.

- The last `itemize` entry contains an `enumerate`:

  1. One.
  2. Two.
  3. Three.

- One.
- Two.
- Three.
- Four.

3.10. Frames and Colors

The function `frame` embeds a text inside a frame. The function `color` may also use the same purpose when it is specified a `bg` option. This is why both functions are included in the same Skribe manual section.

3.10.1. Frame

```
(frame [:border 1] [:margin 2] [:width] [:class "frame"] [:ident] node…)
```

:ident html lout latex context info xml
The node identifier.

:class html lout latex context info xml
The node class.

:width html lout latex context
The 2.1.2.1 of the frame.

:margin html lout latex context

The margin pixel size of the frame.

:border html lout latex context

The border pixel of the frame.

node…

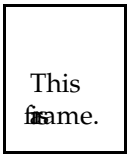
The items of the enumeration.

See also color, p. 36 , table, p. 42 .

Example 13. The frame markup

```
(center (frame :width 10. :margin 10 (p [This is a frame.])))
```

… produces:



3.10.2. Color

The color markup enables changing *locally* the text of the document. If the bg color is used, then, color acts as a container. Otherwise, it acts as an 3.5.

```
(color [:margin] [:width] [:fg] [:bg] [:class "color"] [:ident] node…)
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:width html lout latex context

The 2.1.2.1 of the frame.

:margin html context

The margin pixel size of the frame.

:bg html lout latex context

The background color

:fg html lout latex context

The foreground color

node…

The items of the enumeration.

See also frame, p. 35 , table, p. 42 .

Example 14. The color markup

```
(center
   (color :bg "#aaaaaa"
      :margin 10
      :width 30.
      (center
          (color :bg "#eeeeee" :fg "blue" :width 100. :margin 10 [This is an
example of color box that uses a color for the
background ,(emph "and") the ,(color :fg "red" "foreground"). It also spec-
ifies
a width, that is, an horizontal space, the text should
span to.])))))
```

… produces:

This is an example of color box that uses a color for the background *and* the foreground. It also specifies a width, that is, an horizontal space, the text should span to.

3.11. Figures

The figure markup shown below produces floating figures. Well, whether the figure is really "floating" depends on the engine used (see Chapter 13): printed output as produced by the lout and latex engines do produce floating figures, but on-line output as produced by the html engine does not.

(figure [:multicolumns] [:number #t] [:legend] [:class "figure"] [:ident] body**)**

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:legend html lout latex context info

The legend of the figure. If no **:ident** is provided to the figure, it uses the legend value as an identifier. In consequence, it is possible to use the **:legend** value in references.

:number html lout latex context info

If the optional argument **:number** is a number, that number is used as the new Scribe compiler figure counter. If it is `#t` the compiler automatically sets a number for that figure. If it is `#f` the figure is numberless.

:multicolumns html lout latex context info

A boolean that indicates, for back-ends supporting multi-columns rendering (e.g., "TeX"), if the figure spans over all the columns.

body

The body of the figure.

See also `ref`, p. 56 , `document`, p. 20 .

Example 15. The figure markup

```
(center
 (figure :legend "This is a unnumbered figure"
         :ident "fig1"
         :number #f
         (frame [Skribe is a functional programming language.])))

(center
 (figure :legend "The great Penguin"
         (image :file "linux.png")))
```

Skribe is a functional programming language.

Figure . This is a unnumbered figure

Figure 1. The great Penguin

3.11.1. List of Figures

Skribilo has no built-in facility to display the list of figures. Instead, it provides a general machinery for displaying any kind of lists of items contained in a document. This is described in the section [?section Resolve: ./figure.skb:65:10:] and 1.6.3. For the sake of simplicity, an example showing how to display the list of figures of a document is included below.

Example 16. The figure markup

```
(resolve (lambda (n e env)
            (let* ((d (ast-document n))
                   (ex (container-env-get d 'figure-env)))
               (table (map (lambda (e)
                              (tr (td :align 'left
                                      (markup-option e ':number)
                                      " "
                                      (ref :handle (handle e)
                                           :text (markup-option e :legend))
                                      " (section "
                                      (let ((c (ast-section e)))
                                         (ref :handle (handle c)
                                              :text (markup-option c :ti-
tle)))
                                      ")")))
                           (sort ex
                                 (lambda (e1 e2)
                                    (let ((n1 (markup-option e1 :number))
                                          (n2 (markup-option e2 :number)))
                                       (cond
                                          ((not (number? n1))
                                           #t)
                                          ((not (number? n2))
                                           #f)
                                          (else
                                           (< n1 n2)))))))))))
```

… produces:

This is a unnumbered figure (section Figures)

1 The great Penguin (section Figures)

3.12. Images

Images are defined by the means of the `image` function

```
(image :file [:zoom] [:height] [:width] [:url] [:class] [:ident] comment)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:file` html lout latex context info

The file where the image is stored on the disk (see image path). The image is converted (see `convert-image`) into a format supported by the engine. This option is exclusive with the **:url** option.

`:url` html lout latex context info

The URL of the file. This option is exclusive with the option.

`:width` html lout latex context info

The width of the image. It may be an integer for a pixel size or a floating point number for a percentage.

`:height` html lout latex context info

The height of the image. It may be an integer for a pixel size or a floating point number for a percentage.

`:zoom` lout latex context

A zoom factor.

`comment`

A text describing the image.

See also `*image-path*`, p. 41 , `convert-image`, p. 42 .

Example 17. The image markup

```
(image :file "linux.png" "A first image")
(image :height 50 :file "linux.png" "A smaller one")
(image :file "bsd.png" "A second image")
(image :width 50 :file "bsd.png")
(image :width 200 :height 40 :file "bsd.png")
```

… produces:

Files passed as a `:file` argument to `image` are searched in the current *image path*, which is defined by the `*image-path*` SRFI-39 parameter. This parameter contains a list of directories and its value can be obtained using `(*image-path*)`. Its value can be altered using the `-P` command-line option of the `skribilo` compiler (see Chapter 14 for details).

3.12.1. Image formats

Images are unfortunately *unportable*. The various Skribe output formats support different image formats. For instance, HTML supports `gif` and `jpeg` while the LaTeX back-end only supports `ps`. Skribe tries, only when needed, to automatically *convert* images to a format supported by the target to be produced. For this, it uses external tools. The default Skribe translation scheme is:

- Do not translate an image that needs no conversion.
- Uses the `fig2dev` external tool to translate `Xfig` images.
- Uses the `convert` external tools to translate all other formats.

Engines support different image formats. Each engine may specify a converter to be applied to an image. The engine custom `image-format` specifies the list of supported image formats. This list is composed of a suffix such as `jpeg` or `gif`.

The function `convert-image` tries to convert an image according to a list of formats. All the specified formats are successively tried. On the first success, the function `convert-image` returns the name of the new converted image. On failure, it returns `#f`.

```
(convert-image file formats)
```

`file`

The image file to be converted. The file is searched in the *image-path*image path.

`formats`

A list of formats into which images are converted to.

See also `*image-path*`, p. 41 .

3.13. Table

Tables are defined by the means of the `table` function.

```
(table [:rulecolor] [:cellspacing] [:cellpadding] [:cellstyle 'collapse]
    [:rules 'none] [:frame 'none] [:width] [:border] [:&location] [:class]
    [:ident] row…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:border` html lout context info

The table border thickness.

`:width` html lout latex context info

The 2.1.2.1 of the table.

`:frame` html lout latex context info

Which parts of frame to render. Must be one of `none`, `above`, `below`, `hsides`, `vsides`, `lhs`, `rhs`, `box`, `border`.

`:rules` html lout latex context info

Rulings between rows and cols, Must be one of `none`, `rows`, `cols`, `header`, `all`.

`:cellstyle` html latex

The style of cells border. Must be either `collapse`, `separate`, or a length representing the horizontal and vertical space separating the cells.

:cellpadding html lout context info

A number of pixels around each cell.

:cellspacing html

An optional number of pixels used to separate each cell of the table. A negative uses the target default.

:rulecolor lout info

The color of rules (see Section 3.10).

row…

The rows of the table. Each row must be constructed by the trtr function.

Note: Table rendering may be only partially supported by graphical agents. For instance, the `cellstyle` attribute is only supported by HTML engines supporting CSS2.

3.13.1. Table Row

Table rows are defined by the `tr` function.

```
(tr [:bg] [:class] [:ident] cell…)
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:bg html lout latex context

The background color of the row.

cell…

The row cells.

3.13.2. Table Cell

Two functions define table cells: `th` for header cells and `td` for plain cells.

```
(th [:bg] [:rowspan 1] [:colspan 1] [:valign] [:align 'center] [:width]
    [:class] [:ident] node)
(td [:bg] [:rowspan 1] [:colspan 1] [:valign] [:align 'center] [:width]
    [:class] [:ident] node)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:bg` html lout

The background color of the cell.

`:width` html lout latex context

The 2.1.2.1 of the table.

`:align` html lout latex context

The horizontal alignment of the table cell (`left`, `right`, or `center`. Some engines, such as the HTML engine, also supports a character for the alignment.)

`:valign` html lout latex context

The vertical alignment of the cell. The value can be `top`, `center`, `bottom`.

`:colspan` html lout latex context

The number of columns that the cell expands to.

`:rowspan` html lout

The number of columns that the cell spans over.

`node`

The value of the cell.

3.13.3. Example

Example 18. A table

```
(center
   (table :border 1 :width 50. :frame 'hsides :cellstyle 'collapse
      (tr :bg "#cccccc" (th :align 'center :colspan 3 "A table"))
      (tr (th "Col 1") (th "Col 2") (th "Col 3"))
      (tr (td :align 'center "10") (td "-20") (td "30"))
      (tr (td :align 'right :rowspan 2 :valign 'center "12") (td "21"))
      (tr (td :align 'center :colspan 2 "1234"))
      (tr (td :align 'center :colspan 2 "1234") (td :align 'right "5"))
      (tr (td :align 'center :colspan 1 "1") (td :colspan 2 "2345"))))
```

… produces:

A table		
Col 1	Col 2	Col 3
10	-20	30
12	21	
	1234	
1234		5
1	2345	

3.14. Footnote

By default, footnotes appear at the bottom of the page that contains the reference to the footnote.

```
(footnote [:label #t] [:class "footnote"] [:ident] text…)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:label` html lout info

This may be either a boolean (i.e., #f or #t) indicating whether a footnote label should automatically be produced, a string specifying a label to use (e.g., "*"), or a number.

`text…`

The text of the footnote.

See also `document`, p. 20 , `chapter`, p. 23 , `section`, p. 24 .

Example 19. A footnote

```
[Scheme,(footnote [To be pronounced ,(char "[")Skim,(char "]")])
is a programming language,(footnote [And a great one!]).]
```

… produces:

Scheme[1] is a programming language[2].

3.15. Characters, Strings and Symbols

3.15.1. Characters

The function `char` introduces a *character* in the produced document. The purpose of this function is to introduce escape characters or to introduce characters that cannot be typesetted in the document (for instance because the editors does not support them). The escapes characters are `[`, `]` and `;`.

(char char**)**

`char`

The character to be introduced. Specified value can be a character, a string or an integer

Example 20. Some characters

```
(itemize (item [The character ,(code "#\\a"): ,(char #\a).])
         (item [The character ,(code "\"a\""): ,(char "a").])
         (item [The character ,(code "97"): ,(char 97).]))
```

… produces:

- The character `#\a`: a.
- The character `"a"`: a.
- The character `97`: a.

[1] To be pronounced [Skim]

[2] And a great one!

3.15.2. Strings

the function `!` introduces raw strings in the target. That is, the strings introduced by `!` are written *as is*, without any transformation from the engine.

```
(! format node…)
```

`format`

The format of the command.

`node…`

The arguments.

The sequences `$1`, `$2`, … in the `format` are replaced with the actual values of the arguments `node`.

Example 21. Some characters

```
[A simple ,(! "string"). A more annoying one ,(! "@B { string }").
A last one with arguments ,(! "@Underline { $1 $2 }" (bold 1) (it 2)).]
```

… produces:

A simple string. A more annoying one **string**. A last one with arguments **1** *2*.

3.15.3. Symbols

The function `symbol` introduces special symbols in the produced file. Note that the rendering of symbols is unportable. It depends of the capacity of the targeted format.

```
(symbol symbol)
```

`symbol`

The symbol to introduce.

Here is the list of recognized symbols:

Symbol name	Rendering

–>	⟶
->	→
1/2	½
1/4	¼
3/4	¾
<+	↵
<-	←
<–	⟵
<–>	⟷
<->	↔
<=	⇐
<==	⟸
<==>	⟺
<=>	⇔
==>	⟹
=>	⇒
AEligature	œ
Aacute	Á
Acircumflex	Â
Agrave	À
Alpha	Α
Amul	Ä
Aring	Å
Atilde	Ã
Beta	Β
Ccedilla	Ç
Chi	Χ
Delta	Δ
Downarrow	⇓
ETH	Ð
Eacute	É
Ecircumflex	Ê
Egrave	È
Epsilon	Ε

Eta	Η
Euml	Ë
Gamma	Γ
Iacute	Í
Icircumflex	Î
Igrave	Ì
Iota	Ι
Iuml	Ï
Kappa	Κ
Lambda	Λ
Mu	Μ
Ntilde	Ñ
Nu	Ν
Oacute	Ó
Ocircumflex	Ô
Ocurcumflex	Ocurcumflex
Oeligature	Œ
Ograve	Ò
Omega	Ω
Omicron	Ο
Oslash	ø
Otilde	Õ
Ouml	Ö
Phi	Φ
Pi	Π
Psi	Ψ
Rho	Ρ
Sigma	Σ
THORN	THORN
Tau	Τ
Theta	Θ
Uacute	Ú
Ucircumflex	Û

Ugrave	Ù
Uparrow	⇑
Upsilon	Υ
Uuml	Ü
Xi	Ξ
Yacute	Ý
Zeta	Ζ
aacute	á
acircumflex	â
aeligature	æ
agrave	à
alef	ℵ
alpha	α
amul	ä
and	∧
angle	∠
approx	≈
aring	å
asterisk	∗
atilde	ã
beta	β
bottom	⊥
bullet	•
cap	∩
ccedilla	ç
cent	¢
chi	χ
circ	○
clubs	♣
cong	≅
copyright	©
cup	∪
currency	currency
dag	†

dashv	⊣
ddag	‡
degree	°
delta	δ
diams	♦
divide	÷
downarrow	↓
eacute	é
ecircumflex	ê
egrave	è
ellipsis	…
emptyset	∅
epsilon	ε
equiv	≡
eta	η
eth	eth
euml	ë
euro	€
exists	∃
female	female
forall	∀
gamma	γ
ge	≥
hearts	♥
iacute	í
icircumflex	î
iexcl	¡
igrave	ì
image	ℑ
in	∈
infinity	∞
integral	∫
iota	ι
iquestion	¿

iuml	ï
kappa	κ
lambda	λ
langle	⟨
lceil	⌈
le	≤
lfloor	⌊
lguillemet	«
lhd	◁
loz	◊
male	male
micro	µ
mid	|
middot	·
models	⊨
mu	µ
mul	×
nabla	∇
neq	≠
ni	∋
not	¬
notin	∉
nsupset	nsupset
ntilde	ñ
nu	ν
oacute	ó
ocurcumflex	ô
oeligature	œ
ograve	ò
omega	ω
omicron	o
oplus	⊕
or	∨
oslash	ø

otilde	õ
otimes	⊗
ouml	ö
paragraph	¶
parallel	∥
partial	∂
perp	⊥
phi	ϕ
pi	π
piv	
plusminus	±
pound	£
prod	∏
propto	∝
psi	ψ
rangle	⟩
rceil	⌉
real	ℜ
registered	registered
rfloor	⌋
rguillemet	»
rhd	▷
rho	ρ
section	§
sigma	σ
sigmaf	
sim	~
spades	♠
sqrt	√
subset	⊂
subseteq	⊆
sum	∑
supset	⊃
supseteq	⊇

szlig	ß
tau	τ
therefore	therefore
theta	θ
thetasym	
thorn	thorn
times	×
tm	™
top	⊤
uacute	ú
ucircumflex	û
ugrave	ù
uparrow	↑
upsilon	υ
uuml	ü
vdash	⊢
weierp	℘
xi	ξ
yacute	ý
yen	¥
ymul	ÿ
zeta	ζ

Chapter 4. References and Hyperlinks

Skribilo supports traditional *cross-references* (that is, references to some part of documents) and *hyperlinks* (that is, visual marks enriching texts that enable interactive browsing). Hyperlinks and references may point to:

- Inner parts of a document, such as a section or a figure.
- Other documents, such as Web documents.
- Other Skribe documents.
- Specific part of other Skribe documents, such as a chapter of another Skribe document.

In order to use hyperlinks, Skribilo documents must:

- *Refer to* marks. This is the role of the `ref` Skribe function.
- *Set* marks. This is the role of the `mark` function. However, most Skribe functions that introduce text structures (e.g., chapters, sections, figures, …) automatically introduce marks as well. So, it is useless to *explicitly* introduce a mark at the beginning of these constructions in order to refer to them with an hyperlink.

4.1. Mark

The `mark` function sets a mark in the produced document that can be referenced to with the `ref` function. Unless a **:text** option is specified, no visible text in associated with the mark in the generated document.

```
(mark [:text] [:class "mark"] [:ident] mark)
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:text

A text associated with the markup.

`mark`

A string that will be used in a ref function call to point to that mark.

The Skribe functions `chapter`, `section`, `subsection`, `subsubsection` Skribe automatically set a mark whose value is the title of the section. The Skribe function `figure` automatically sets a mark whose value is the legend of the figure.

4.2. Reference

Skribilo proposes a single function that can be used for most references. This same `ref` function is used for introducing references to section, to bibliographic entries, to source code line number, etc.

```
(ref [:sort-bib-refs bib-sort-refs/number] [:page] [:skribe] [:line]
    [:handle] [:mark] [:figure] [:url] [:bib-table (*bib-table*)] [:bib]
    [:subsubsection] [:subsection] [:section] [:chapter] [:text] [:ident]
    [:class])
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:text html lout latex context info

The text that is the visual part the links for engines that support hyperlinks.

:url html lout latex context info xml

An URL, that is, a location of another file, such as an HTML file.

:mark html lout latex context info

A string that is the name of a mark. That mark has been introduced by a mark markup.

:handle html lout latex context info

A Skribe node handle.

:ident html lout latex context info xml

The identifier of a node (which was specified as an value).

:figure html lout latex context info

The identifier of a figure.

:chapter html lout latex context info

The title of a chapter.

`:section` html lout latex context info

The title of a section.

`:subsection` html lout latex context info

The title of a subsection.

`:subsubsection` html lout latex context info

The title of a subsubsection.

`:page` lout latex context info

A boolean enabling/disabling page reference (for hard copies as produced by the Lout and LaTeX engines).

`:bib` html lout latex context info xml

A name or a list of names of bibliographic entry.

`:bib-table`

The bibliography where searching the entry.

`:sort-bib-refs`

In case where multiple bibliography entries are referenced, as in `(ref :bib '("smith81:disintegration" "corgan07:zeitgeist"))`, this should be a two-argument procedure suitable for sorting. The default procedure sorts references by number, when `the-bibliography` uses the `number` labeling style. If it is `#f`, then references will not be sorted.

`:line` html lout latex context info xml

A reference to a progprogram line number.

`:skribe` html lout latex context info xml

The name of a Skribe Url IndexSkribe Url Index file that contains the reference. The reference can be a chapter, section, subsection, subsubsection or even a mark located in the Skribe document described by the file sui.

See also `index`, p. 62 , `numref`, p. 57 , `the-bibliography`, p. 68 .

Sometimes, it is useful to produce phrases that refer a section by its number, as in "See Section 2.3". This is especially useful on printed documents, as produced by the Lout and LaTeX engines. The `numref` markup is provided to that end:

```
(numref [:class] [:separator "."] [:page] [:text ""] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:text`

Text preceding the reference number.

`:ident` html lout latex context info xml

The identifier of the node (a chapter, section, subsection, etc.) being referenced.

`:page`

A boolean enabling/disabling page reference (for hard copies as produced by the Lout and LaTeX engines).

`:separator`

The separator between numbers.

See also `ref`, p. 56 .

Example 22. Some references

```
[This hyperlink points to the ,(ref :figure "The great Penguin" :text
"figure")
of the chapter ,(ref :chapter "Standard Markups") (or also, the
,(ref :ident "std-markups" :text "chapter") about markups).
In the second example of reference, no ,(code ":text") option is specified:
,(ref :figure "The great Penguin"). One may use the ,(param ":ident")
field when specified such as: ,(ref :ident "fig1") or ,(ref :figure "fig1").

,(linebreak)
That other one points to a well known
,(ref :url "http://slashdot.org/" :text "url"). The same without
,(code ":text"): ,(ref :url "http://slashdot.org/").

,(linebreak)
And one can also refer to sections by number, as in "see ,(numref :text
[Wonderful Section] :ident "refs")".

,(linebreak)
With more complex tricks that are explained in Section
,(ref :section "Resolve"), it is also possible use, for the text of the
reference, a container number such as chapter:
,(resolve (lambda (n e env)
              (let ((s (find1-down (lambda (x)
                                     (and (is-markup? x 'chapter)
                                          (string=? (markup-option x :ti-
tle)
                                                    "Standard Markups")))
                                   (ast-document n))))
                (ref :handle (handle s) :text (markup-option s :num-
ber))))).]
```

… produces:

This hyperlink points to the figure of the chapter 3 (or also, the chapter about markups). In the second example of reference, no `:text` option is specified: 1. One may use the :ident field when specified such as:

That other one points to a well known url. The same without `:text`: http://slashdot.org/.

And one can also refer to sections by number, as in "see Wonderful Section 4.2".

With more complex tricks that are explained in Section [**?section Resolve: ./src/links1.skb:19:2:**], it is also possible use, for the text of the reference, a container number such as chapter: 3.

4.3. Electronic Mail

The `mailto` function is mainly useful for electronic output formats that are able to run a mailing agent. The function `mailto` introduces mail annotation in a Skribe document.

```
(mailto :text [:class "mailto"] [:ident] email)
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:text` html lout latex context info

The text that is the visual part the links.

`email`

The electronic address.

Example 23. Mail address reference

```
[It is possible to send a mail by
,(mailto "foo@nowhere.com" :text "clicking") that link. That same
reference without ,(code ":text") options: ,(mailto "foo@nowhere.com").
]
```

… produces:

It is possible to send a mail by *clicking* that link. That same reference without `:text` options: *foo@nowhere.com*.

4.4. Skribe URL Index

A *Skribe URL Index* (henceforth SUI) describes the marks that are available in a Skribe or Skribilo document. It is to be used to make marks available to other Skribe/Skribilo documents through the **:skribe** option of the `ref` markup. The syntax of a SUI file is:

```
<sui>     –> (skribe-url-index <title>
              :file <file-name>
              (marks <sui-ref>*)
              (chapters <sui-ref>*)
              (section <sui-ref>*)
              (subsection <sui-ref>*)
              (subsubsection <sui-ref>*))
<sui-ref> –> (<string> :file <file-name> :mark <string>)
```

SUI files can be automatically produced by the Skribilo compiler. For instance, in order to produce the SUI file of this user manual, one should set the `emit-sui` HTML custom to `#t`; a `user.sui` file will then be produced when the manual is compiled to HTML:

```
skribilo -t html -o user.html user.skb
```

Chapter 5. Indexes

Skribe support indexes. One may accumulate all entries inside one unique index or dispatch them amongst user declared indexes. Indexes may be *monolithic* or *split*. They only differ in the way they are rendered by the back-ends. For a split index a sectioning based on the specific (e.g., "the first one") character of index entries is deployed.

5.1. Making indexes

The function `make-index` declares a new index.

`(make-index` `ident`**`)`**

`ident`

A string, the name the index (currently unused).

See also `default-index`, p. 61 , `index`, p. 62 , `the-index`, p. 63 , `ref`, p. 56 , `mark`, p. 55 .

For instance, the following Skribe expression declares an index named `*index1*`:

Example 24. Creation of a new index

```
(define *index1* (make-index "a new index"))
```

This example produces no output but enables entries to be added to that index. In general it is convenient to declare indexes *before* the call to the `document` function.

The function `default-index` returns the default index that pre-exists to all execution.

`(default-index)`

5.2. Adding entries to an index

The function `index` adds a new entry into one existing index and sets a mark in the text where the index will point to. It is an error to add an entry into an index that is not already declared.

```
(index [:url] [:shape] [:index] [:note] [:class "index"] [:ident] name)
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:index

The name of the index whose index entry belongs to. A value of #f means that the default-index owns this entry.

:note

An optional note added to the index entry. This note will be displayed in the index printing.

:shape

An optional shape to be used for rendering the entry.

:url

An optional URL that is referenced in the index table instead of the location of the index.

`name`

The name of the entry. This must be a string.

See also `make-index`, p. 61 , `default-index`, p. 61 , `the-index`, p. 63 .

The following expressions add entries to the index `*index1*`:

Example 25. Adding entries to an index

The identifier `Foo` is a usually used as an example. When two identifiers have to used,

```
[The identifier ,(code "Foo"),(index :index *index1* "Foo") is a usually
used as an example. When two identifiers have to used, frequently the
second choice is ,(code "Bar"),(index :index *index1* "Bar" :shape (it
"Bar")).
When three are needed, some use ,(code "Baz")
,(index :index *index1* "Baz" :shape (it "Baz")).

This illustrates how to use identifier
,(index :index *index1* "Foo" :note "How to use Foo")
,(index :index *index1* "Foo" :note "How not to use Foo")
```

```
,(index :index *index1* "Fooz")
…]
```

frequently the second choice is `Bar`. When three are needed, some use `Baz`. This illustrates how to use identifier …

There is no output associated with these expressions.

5.3. Printing indexes

The function `the-index` displays indexes in the produced document.

(the-index [**:column** 1] [**:header-limit** 50] [**:char-offset** 0] [**:split**] [**:class** "the-index"] [**:ident**] index…**)**

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:split

If #t, character based sectioning is deployed. Otherwise all the index entries are displayed one next to the other.

:char-offset

The character number to use when split is required. This option may be useful when printing index whose items share a common prefix. The argument can be used to skip this prefix.

:header-limit

The number of entries from which an index header is introduced.

:column

The number of columns of the index.

index…

The indexes to be displayed. If index is provided, the global index default-index is printed.

If the engine custom `index-page-ref` is true when a index is rendered then, page reference framework is used instead of a direct reference framework.

Example 26. Printing indexes

```
(the-index *index1*)
```

… produces:

Bar
Baz
Foo
How to use Foo
How not to use Foo
Fooz

See the Skribe [**?mark global index: ./index.skb:127:21:**] for a real life index example.

Chapter 6. Bibliographies

Skribilo provides support for bibliographies. To setup a bibliography database and to be able to refer to its entries from a document, the following things must be done:

- Use the default pre-existing *bibliography table* or create a custom one.
- Provide a *bibliography database*.
- Load the database using the `bibliography` function.
- Reference to bibliography entries with `ref :bib` function calls.

The following sections will guide you through these steps.

6.1. Bibliography Tables

This section describes functions dealing with *bibliography tables*. Essentially, bibliography tables are the representation of your bibliographies used by Skribilo at run-time.

The predicate `bib-table?` returns `#t` if and only if its argument is a bibliography table as returned by `make-bib-table` or `*bib-table*`. Otherwise `bib-table?` returns `#f`.

(bib-table? `obj`**)**

`obj`

The value to be tested

See also `make-bib-table`, p. 65 , `*bib-table*`, p. 65 , `bibliography`, p. 66 , `the-bibliography`, p. 68 .

The function `*bib-table*` returns a global, pre-existing bibliography-table:

(*bib-table*)

See also `bib-table?`, p. 65 , `make-bib-table`, p. 65 , `bibliography`, p. 66 , `the-bibliography`, p. 68 .

Technically, `*bib-table*` is actually an SRFI-39 parameter object, so it can be queried and modified like any other parameter object.

The function `make-bib-table` constructs a new bibliography-table:

(make-bib-table ident**)**

ident

The name of the bibliography table.

See also bib-table?, p. 65 , *bib-table*, p. 65 , bibliography, p. 66 , the-bibliography, p. 68 .

6.2. Bibliography

The bibliography function loads bibliography entries into the bibliography table specified using the **:bib-table** parameter. It can be passed either lists representing entries (such as an article or book reference), or strings denoting the names of files that contains several entries. All the entries loaded in memory can then be referred to with ref. A bibliography database must be loaded *before* any reference is introduced. It is advised to place the bibliography function call before the call to the document function call.

(bibliography [**:bib-table** (*bib-table*)] [**:command**] entry…**)**

:command html lout latex context info xml

An external command to be applied when loading the bibliography entries. The sequence ~a is replaced with the name of the file when the command is invoked.

:bib-table html lout latex context info xml

The table where entry is searched.

entry…

If entry is a string, it denotes a file containing the entry (see bibliograph path). Otherwise, it is a list described by the syntax below.

See also bib-table?, p. 65 , make-bib-table, p. 65 , *bib-table*, p. 65 , the-bibliography, p. 68 .

Files passed as an argument to bibliography are searched in the current *bibliography path*, which is defined by the *bib-path* SRFI-39 parameter. This parameter contains a list of directories and its value can be obtained using (*bib-path*). Its value can be altered using the -B command-line option of the skribilo compiler (see Chapter 14 for details).

The **:command** option can be used to import foreign bibliography. The following example, shows how to directly use a BibTeX bibliography using the 6.4 translator.

Example 27. Printing a Bibliography

```
(bibliography :command "gzip -d -to-stdout ~a | skribebibtex"
"scheme.bib.gz")
```

6.2.1. Bibliography Syntax

The Skribe/Skribilo bibliography database uses a format very close to the BibTeX one, which is a parenthetic version of BibTeX. Here is the syntax of an entry:

<entry> –> **(**<kind> <key> <field>+**)**
<kind> –> techreport | article | inproceedings | book
<key> –> <symbol> | <string>
<field> –> **(**<symbol> <string>**)**

BibTeX files cannot be directly loaded but the tool `skribebibtex` can be use to automatically convert BibTeX format to Skribe bibliography format. Here is an example of a simple Skribe database.

```
(book queinnec:lisp
   (author "Christian Queinnec")
   (title "Lisp In Small Pieces")
   (publisher "Cambridge University Press")
   (year "1996"))

(book scheme:ieee
   (title "IEEE Standard for the Scheme Programming Language")
   (author (noabbrev "IEEE Std 1178-1990"))
   (publisher "Institute of Electrical and Electronic Engineers,
Inc.")
   (address "New York, NY")
   (year "1991"))

(misc bigloo
   (author "Manuel Serrano")
   (year "2006")
   (url "http://www.inria.fr/mimosa/fp/Bigloo"))

(misc scheme:r4rs
   (title [The Revised,(sup [4]) Report on the Algorithmic Language
Scheme])
   (author "William D. Clinger, Jonathan Rees")
   (month "Nov")
   (year "1991")
```

```
   (url
"http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_toc.html"))

(article scheme:r5rs
   (title "The Revised5 Report on the Algorithmic Language Scheme")
   (author "Richard Kelsey, William D. Clinger, Jonathan Rees")
   (journal "Higher-Order and Symbolic Computation")
   (volume "11")
   (number "1")
   (month "Sep")
   (year "1998")
   (url "http://kaolin.unice.fr/Bigloo/doc/r5rs.html"))

(book as:sicp
   (author "Harold Abelson, Gerald Jay Sussman")
   (title "Structure and Interpretation of Computer Programs")
   (year "1985")
   (publisher "MIT Press")
   (address "Cambridge, Mass., USA"))
```

6.3. Printing a Bibliography

The function `the-bibliography` displays the bibliography.

```
(the-bibliography :pred [:labels 'number] [:count 'partial] [:sort
    bib-sort/authors] [:bib-table (*bib-table*)])
```

`:bib-table` html lout latex context info xml

The bibliography table to be displayed.

`:pred` html lout latex context info xml

A predicate filtering the bibliography entries. It takes two parameters: the bibliography entry and the `the-bibliography` node.

`:sort` html lout latex context info xml

A function sorting a list of entries.

`:labels` html lout latex context info xml

Specifies the style for bibliography entries labels. The default, `number`, uses numbers to identify references, e.g., "[7]". When `name+year` is chosen, long labels including the first author's last name (and optionally the second author's last name) and the year of publication will be used. For instance: "[Smith 1984]", "[Smith & Johnson 1979]", or "[Smith *et al.* 1980]".

`:count` html lout latex context info xml

The symbol `partial` or `full` specifies the numbering to be applied. The value `partial` tells Skribilo to count only the entries filtered in by **:pred**. The value `full` tells Skribilo to count all entries, event those filtered out by **:pred**.

See also `bib-table?`, p. 65 , `make-bib-table`, p. 65 , `*bib-table*`, p. 65 , `bibliography`, p. 66 , `noabbrev`, p. 69 .

Note that the `name+year` label style will only work if the following conventions are followed by the `author` field of your bibliography entries:

- the `author` fields of bibliographic entries should be a string containing a comma-separated list of full author names;
- each "full author name" should have the form `first-name second-name … last-name`.

When using the `name+year` label style, it is sometimes desirable to preclude automatic abbreviations for some authors, e.g., when the author is the name of a consortium or company rather than that of a person. In that case, you should enclose the value of your `author` field in a `noabbrev` markup.

(noabbrev `[:class]` `[:ident]`**)**

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

See also `the-bibliography`, p. 68 , `bib-sort/first-author-last-name`, p. 73 .

The following example illustrates typical use of a bibliography.

Example 28. Printing a Bibliography

```
[Scheme ,(ref :bib 'scheme:r5rs) is functional programming language. It
exists
several books about this language ,(ref :bib '(as:sicp queinnec:lisp)).

,(linebreak 2)
,(center (bold [- Bibliography -]))

,(center (frame :border 1 :margin 2 :width 90. (the-bibliography)))]
```

… produces:

Scheme [5] is functional programming language. It exists several books about this language [2, 3].

– Bibliography –

[2] Christian Queinnec. Lisp In Small Pieces. Cambridge University Press, 1996.

[3] Harold Abelson, Gerald Jay Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass., USA, 1985.

[5] Richard Kelsey, William D. Clinger, Jonathan Rees. The Revised5 Report on the Algorithmic Language Scheme. In *Higher-Order and Symbolic Computation*, 11(1) , Sep 1998, .

Note that the current locale setting affects the language used in bibliography entries. For instance, if the `LC_ALL` environment variable is set to `sv_SE.utf8`, then phrases like "Chapter" or "Technical Report" will be written in Swedish.

6.3.1. Filtering Bibliography Entries

The **:pred** option is bound to a function of one argument that filters bibliography entries. It is used to control which entries must appears on a bibliography. The default behavior is to display only the entries referenced to in the text. For instance, in order to display *all* the entries of a bibliography, is it needed to print the bibliography with a predicate returning always `#t`.

Example 29. Unfiltering Bibliography Entries

```
(center
 (frame :border 1 :margin 2 :width 90.
        (the-bibliography :pred (lambda (m n) #t))))
```

… produces:

[1] IEEE Std 1178-1990. IEEE Standard for the Scheme Programming Language. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[2] Christian Queinnec. Lisp In Small Pieces. Cambridge University Press, 1996.

[3] Harold Abelson, Gerald Jay Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass., USA, 1985.

[4] Manuel Serrano. *http://www.inria.fr/mimosa/fp/Bigloo*. 2006. *http://www.inria.fr/mimosa/fp/Bigloo*.

[5] Richard Kelsey, William D. Clinger, Jonathan Rees. The Revised5 Report on the Algorithmic Language Scheme. In *Higher-Order and Symbolic Computation*, 11(1) , Sep 1998, .

[6] William D. Clinger, Jonathan Rees. The Revised4 Report on the Algorithmic Language Scheme. Nov 1991. *http://www.cs.indiana.edu/scheme-repository/R4RS/r4rs_toc.html*.

The second example, filters out the entries that are not *book* or that are not referenced to from the document.

Example 30. Unfiltering Bibliography Entries

```
(center
 (frame :border 1 :margin 2 :width 90.
        (the-bibliography :pred (lambda (m n)
                                  (and (eq? (markup-option m 'kind) 'book)
                                       (pair? (markup-option m
'used)))))))
```

… produces:

[2] Christian Queinnec. Lisp In Small Pieces. Cambridge University Press, 1996.

[3] Harold Abelson, Gerald Jay Sussman. Structure and Interpretation of Computer Programs. MIT Press, Cambridge, Mass., USA, 1985.

The last example, illustrates how to change the rendering of a bibliography. It uses the [?mark processor: skribilo/documentation/manual.scm:307:4:] construction and it defines two [?ident writer: ./bib.skb:252:21:] for displaying `&bib-entry-ident` and `&bib-entry-title` markups. These two markups are introduced by Skribe when it loads a bibliography. All fields of bibliography entries are represented by markups whose prefix are `&bib-entry-`. The parent of all these markups is the bibliography entry itself. The `&bib-entry-` markups are options of there parent.

Example 31. Unfiltering Bibliography Entries

```
(center
 (frame :border 1 :margin 2 :width 90.
         (processor :engine
                    (make-engine '_ :filter string-upcase)
                    :combinator
                    (lambda (e1 e2)
                       (let ((e (copy-engine '_ e2)))
                          (markup-writer '&bib-entry-ident e
                                         :action
                                         (lambda (n e)
                                            (let* ((be (ast-parent n))
                                                   (o (markup-option be
'author))
                                                   (y (markup-option
be 'year)))
                                               (output (markup-body o) e1)
                                               (display ":")
                                               (output (markup-body y) e))))
                          (markup-writer '&bib-entry-title e
                                         :action
                                         (lambda (n e)
                                           (evaluate-document (it (markup-body
n)) e)))
                          e))
                    (the-bibliography :pred
                                      (lambda (m n)
                                         (eq? (markup-option m 'kind)
'book))))))
```

... produces:

[IEEE Std 1178-1990] *IEEE Standard for the Scheme Programming Language*. Institute of Electrical and Electronic Engineers, Inc., New York, NY, 1991.

[CRETIN 1996] *Christian Queinnec. Lisp in Small Pieces*. Cambridge University Press, 1996.

[HAR 1985] *Harold Abelson, Gerald Jay Sussman. Structure and Interpretation of Computer Programs*. MIT Press, Cambridge, Mass., USA, 1985.

6.3.2. Sorting Bibliography Entries

The **:sort** option of the `the-bibliography` markup is used for sorting the bibliography entries. There are three pre-existing functions for sorting entries, defines in the `(skribilo biblio)` module (see Section 1.5).

(bib-sort/authors `l`**)**

(bib-sort/idents `l`**)**

(bib-sort/dates `l`**)**

`l`

The list of entries.

See also `bib-sort/first-author-last-name`, p. 73 .

The first function sorts the entries according to an alphabetic ordering on authors. The second sorts according to an alphabetic ordering on entries identifier. The last one sorts according to entries date.

Example 32. Sorting Bibliography Entries

```
(define (bib-sort/idents l)
   (sort l (lambda (e f) (string<? (markup-ident e) (markup-ident f)))))
```

In addition, the `(skribilo biblio author)` module exports a fourth procedure that sorts bibliography entries:

(bib-sort/first-author-last-name `entries`**)**

`entries`

The list of entries.

See also `bib-sort/authors`, p. 73 , `the-bibliography`, p. 68 , `noabbrev`, p. 69 .

This procedure allows entries to be sorted according to the last name of the first author of each bibliography entry. For this to work, the author field of bibliography entries must follow the same conventions as for the `name+year` label style of `the-bibliography`.

6.4. Skribebibtex

FIXME: This tool is not available as of Skribilo version 0.9.3.

Chapter 7. Computer Programs

In a document describing a computer programming language, it is common to include excerpt of programs. Program code is typically typeset in a specific font, with no justification, and with a precise *indentation*. Indentation is important because it helps understand the code; it is thus desirable to preserve indentation in program text. The `pre` text layout already allows indentation to be preserved. This chapter presents two new functions that complement it: `prog` and `source`, both of which are specially designed to represent computer programs in text.

7.1. Program

A `prog` function call preserves the indentation of the program. It may automatically introduce line numbers.

```
(prog [:mark ";!"] [:linedigit] [:line 1] [:class "prog"] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:line` html lout latex context info xml

Enables/disables automatic line numbering. An integer value enables the line number and specifies the number of the first line of the program. A value of `#f` disables the line numbering.

`:linedigit` html lout latex context info xml

The number of digit for representing line numbers.

`:mark` html lout latex context info xml

A string or the boolean `#f`. If this option is a string, that string is the prefix of line marks. These marks can be used in the `ref` reference. A mark identifier is defined by the regular expression: `[_a-zA-Z][_a-zA-Z0-9]*`. The prefix and the mark are removed from the output program.

See also `source`, p. 76 , `pre`, p. 32 , `ref`, p. 56 .

Example 33. A program

```
(frame :width 100.
       (prog :line 10 :mark "##" [
SKRIBILO = skribilo

all: demo.html demo.lout  ##main-goal

demo.html: demo.skb
         $(SKRIBILO) -t html demo.skb -o demo.html

demo.lout: demo.skb
         $(SKRIBILO) -t lout demo.skb -o demo.lout
]))

(p [The main goal is specified on line ,(ref :line "main-goal").])
```

… produces:

```
10.     SKRIBILO = skribilo
11.
12.     all: demo.html demo.lout
13.
14.     demo.html: demo.skb
15.         $(SKRIBILO) -t html demo.skb -o demo.html
16.
17.     demo.lout: demo.skb
18.         $(SKRIBILO) -t lout demo.skb -o demo.lout
```

The main goal is specified on line 12.

7.2. Source Code

The `source` function extracts part of the source code and enables *fontification*. That is, some words of the program can be rendered using different colors or faces.

```
(source :language [:tab 8] [:definition] [:stop] [:start] [:file])
```

`:language` html lout latex context info xml

The `language` of the source code.

`:file` html lout latex context info xml

The file containing the actual source code. The file is searched in the `*source-path*` path.

`:start` html lout latex context info xml

A start line number or a start marker.

:stop html lout latex context info xml

A stop line number or a stop marker.

:definition html lout latex context info xml

The identifier of the definition to extract.

:tab html lout latex context info xml

The tabulation width.

See also prog, p. 75 , language, p. 80 , ref, p. 56 , *source-path*, p. 79 .

Example 34. The source markup

```
(use-modules (skribilo source lisp))

(linebreak)
(frame :width 100.
       (prog (source :language scheme :file "prgm.skb" :definition 'fib)))

(p [The Fibonacci function is defined on line ,(ref :line "fib").])
(linebreak)


(frame :width 100.
       (prog :line 23 :mark #f
             (source :language skribe :file "prgm.skb" :start 22 :stop
29)))


(p [Here is the source of the frame above:])
(linebreak)

(frame :width 100.
       (prog :line 30 :mark #f
             (source :language skribe :file "src/prgm2.skb"
                :start (string-append ";" "!start") ;; trick!
                :stop (string-append ";" "!stop"))))
```

… produces:

```
1.      (define (fib x)
2.        (if (< x 2)
3.            1
4.            (+ (fib (- x 1)) (fib (- x 2)))))
```

The Fibonacci function is defined on line 1.

```
23.     ;*---------------------------------------------------------------------*/
24.     ;*    fib ...                                                          */
25.     ;*---------------------------------------------------------------------*/
26.     (define (fib x) ;!fib
27.       (if (< x 2)
28.           1
29.           (+ (fib (- x 1)) (fib (- x 2)))))
30.
```

Here is the source of the frame above:

```
30.     (frame :width 100.
31.         (prog :line 23 :mark #f
32.             (source :language skribe :file "prgm.skb" :start 22 :stop 29)))
```

Note that even awful programming languages of the C family can be highlighted!

Example 35. The source markup for C

```
(use-modules (skribilo source c))

(p [Here's how:])

 (linebreak)
 (prog
   (source :language c
[#include <stdlib.h>

static int foo = 10;
static float bar;

/* This is the function responsible
   for integer chbouibification.  */
float
chbouibify (int x)
{
  bar = foo + (float) x / random ();
  foo = (float) x * random ();

  if (x > 2)
    /* Great!  */
    printf ("hello world!\n");
  else
    printf ("lower than two\n");
```

```
  return ((float) foo * bar);
}]))
```

… produces:

Here's how:

```
1.  #include <stdlib.h>
2.
3.  static int foo = 10;
4.  static float bar;
5.
6.  /* This is the function responsible
7.  for integer chbouibification.  */
8.  float
9.  chbouibify (int x)
10. {
11.   bar = foo + (float) x / random ();
12.   foo = (float) x * random ();
13.
14.   if (x > 2)
15.     /* Great!  */
16.     printf ("hello world!n");
17.   else
18.     printf ("lower than twon");
19.
20.   return ((float) foo * bar);
21. }
```

You would highlight Java™ code in a similar way, i.e., with `:language java`.

Files passed as the `:file` argument of `source` are searched in the current *source path*, which is defined by the `*source-path*` SRFI-39 parameter. This parameter contains a list of directories and its value can be obtained using `(*source-path*)`. Its value can be altered using the `-S` command-line option of the `skribilo` compiler (see Chapter 14 for details).

The **:language** parameter of `source` takes a `language` object, which performs the actual source highlighting. Several programming languages are currently supported: the `(skribilo source lisp)` module provides `skribe`, `scheme`, `stklos`, `bigloo` and `lisp`, which implement source highlighting for the corresponding lispy dialects, while the `(skribilo source c)` module provides `c` and `java`. Thus, you need to import the relevant module to get the right language, for instance by adding `(use-modules (skribilo source c))` at the beginning of your document. Additional languages can be created using the `language` function (see below).

7.3. Language

The `language` function builds a language that can be used in `source` function call.

```
(language :name [:extractor] [:fontifier])
```

`:name` html lout latex context info xml

A string which denotes the name of the language.

`:fontifier` html lout latex context info xml

A function of one argument (a string), that colorizes a line source code.

`:extractor` html lout latex context info xml

A function of three arguments: an input port, an identifier, a tabulation size. This function scans in the input port the definition is looks for.

See also `prog`, p. 75 , `source`, p. 76 , `ref`, p. 56 .

Chapter 8. Equation Formatting

Skribilo comes with an equation formatting package. This package may be loaded by adding the following form at the top of your document:

1. (use-modules (skribilo package eq))

It allows the inclusion of (complex) equations in your documents, such as, for example, the following:

$$A\left(D\right) = \sum_{i=b}^{Sb} C_i^{Sb}\left(\mu^{i}\left(1-\mu\right)^{Sb-i}\right)$$

This chapter will describe the syntactic facilities available to describe equations, as well as the rendering options.

8.1. Syntax

To start with, let's have a look at a concrete example.

Example 36. Example of a simple equation using the verbose syntax

```
;; The golden ratio, phi.
(eq (eq:= (symbol "phi")
          (eq:/ (eq:+ 1 (eq:sqrt 5))
                2)))
```

… produces:

$$\phi = \frac{1+\sqrt{(5)}}{2}$$

In this example, the `eq:` sub-markups are used pretty much like any other kind of markup. However, the resulting syntax is very verbose and hard to read.

Fortunately, the `eq` package allows for the use of a much simpler syntax.

Example 37. Example of a simple equation

```
;; The golden ratio, phi.
(eq '(= phi (/ (+ 1 (sqrt 5)) 2)))
```

… produces:

$$\phi = \frac{1+\sqrt{(5)}}{2}$$

Readers familiar with the Lisp family of programming languages may have already recognized its *prefix notation*. Note that, unlike in the previous example, the equation itself if *quoted*, that is, preceded by the ' sign. Additionally, when referring to a symbol (such as the Greek letter ϕ), you no longer need to use the `symbol` markup (see Section 3.15.3).

It is possible to create *equation display blocks*, where several equations are displayed and aligned according to a particular operator.

Example 38. Inlined, displayed, and aligned equations

```
(p [This paragraph contains this equation: ,(eq :inline? #t '(/ alpha
beta)).  This is actually an ,(emph [inline]) equation, meaning that it
occurs within a paragraph.  Typesetting has to be adjusted
accordingly.])

(eq-display

   (p [This is an equation display block, within which equations can be
aligned with one another.])

   (eq :ident "eq-limit-b-over-l"
       :renderer (and %have-lout? 'lout)
       :align-with '=
       '(= (limit (/ lambda beta) 0
                  (apply IPL n k))

            ;; non-simplified
            (/ (expt (+ alpha beta) k)
               (* beta
                  (sum :from (= x 0)
                       :to   (- k 1)
                       (* (combinations k x)
                          (expt beta (- k 1 x))
                          (expt alpha x)))))))

   [This equation can be simplified as follows:]

   (eq :ident "eq-limit-b-over-l-simplified"
       :renderer (and %have-lout? 'lout)
       :align-with '=
       '(= ;; simplified
            (/ (expt (+ alpha beta) k)
               (- (expt (+ alpha beta) k)
                  (expt alpha k)))

            (limit (/ lambda beta) 0
                   (apply IPL n k)))))
```

… produces:

This paragraph contains this equation: $\frac{\alpha}{\beta}$. This is actually an *inline* equation, meaning that it occurs within a paragraph. Typesetting has to be adjusted accordingly.

This is an equation display block, within which equations can be aligned with one another.

$$\lim atop_{\beta}^{\lambda} \to 0\left(IPL\left(n,k\right)\right) = \frac{\left(\alpha+\beta\right)^{k}}{\beta\left(\sum_{x=0}^{k-1}\binom{k}{x}\beta^{k-1-x}\ \alpha^{x}\right)} \tag{8.1}$$

This equation can be simplified as follows:

$$\frac{\left(\alpha+\beta\right)^{k}}{\left(\alpha+\beta\right)^{k}-\alpha^{k}} = \lim atop_{\beta}^{\lambda} \to 0\left(IPL\left(n,k\right)\right) \tag{8.2}$$

8.2. Rendering

8.3. Summary

The options available for the top-level `eq` markup are summarized here:

```
(eq [:number #t] [:mul-style 'space] [:div-style 'over] [:renderer]
    [:align-with] [:inline?  'auto] [:class "eq"] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:inline?` lout

If `auto`, Skribilo will automatically determine whether the equation is to be "in-line". Otherwise, it should be a boolean indicating whether the equation is to appear "in-line", i.e., within a paragraph. If the engine supports it, it may adjust various parameters such as in-equation spacing accordingly.

:number lout

If true, then a number is automatically assigned to the equation and displayed. If it is a string, then that string is used as the equation's number. If `#f`, then the equation is left unnumbered. Note that this option is only taken into account for displayed equations.

:renderer

The engine that should be used to render the equation. This allows, for instance, to use the Lout engine to render equations in HTML.

:mul-style lout

A symbol denoting the default style for multiplications. This should be one of `space`, `cross`, `asterisk` or `dot`.

:div-style lout

A symbol denoting the default style for divisions. This should be one of `over`, `fraction`, `div` and `slash`. Per-`eq`: `/` :`div-style` options override this setting.

:align-with lout

Within a `eq-display` block, this should be a symbol specifying according to which operator equations are to be aligned with one another.

Equation display blocks can be defined using `eq-display`. Display blocks define the scope of the alignment among equations as specified by the :`align-with` options of `eq`.

(eq-display [:**class** "eq-display"] [:**ident**]**)**

:ident html lout latex context info xml

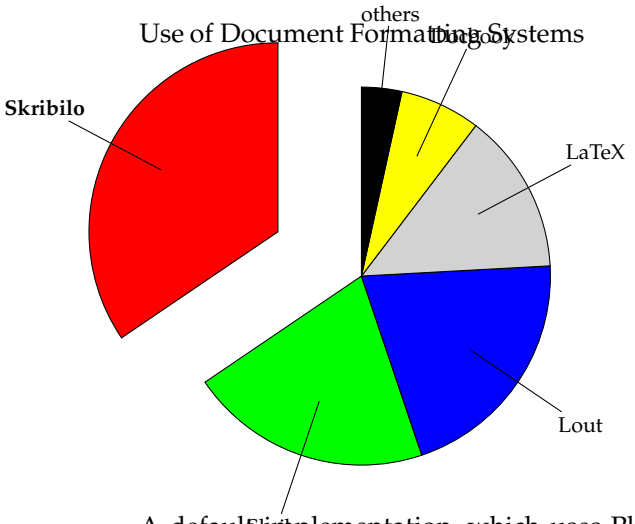
The node identifier.

:class html lout latex context info xml

The node class.

Chapter 9. Pie Charts

Skribilo contains a pie-chart formatting package, located in the `(skribilo package pie)` module. It allows users to produces represent numeric data as pie charts as in the following example:



A default implementation, which uses Ploticus as an external program, is available for all engines. There is also a specific implementation for the Lout engine which relies on Lout's own pie-chart package. In the latter case, you don't need to have Ploticus installed, but you need it in the former.

Currently it only supports slice-coloring, but support for textures (particularly useful for black & white printouts) could be added in the future.

9.1. Syntax

Let us start with a simple example:

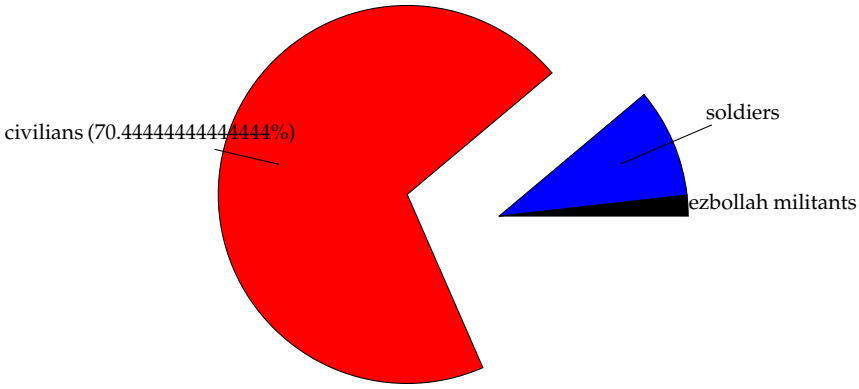
Example 39. Example of a pie chart

```
;; A sad pie chart.
;;

(pie :title [Casualties in the Israel-Lebanon 2006 Conflict (source:
English Wikipedia page, 2006-07-23)]
     :total 450 ;; to show the uncertainty on figures
     :ident "pie-lebanon-2006"
```

```
      :labels 'outside :fingers? #t

  (slice :weight   8 :color "black" [Hezbollah militants])
  (slice :weight  42 :color "blue"  [soldiers])
  (slice :weight 317 :color "red" :detach? #t
         [civilians (,(sliceweight :percentage? #t)%)]))
```

… produces:

Casualties in the Israel-Lebanon 2006 Conflict (source: English Wikipedia page, 2006-07-23)



This illustrates the three markups provided by the `pie` package, namely `pie`, `slice`, and `sliceweight`. This last markup returns the weight of the slice it is used in, be it as a percentage or an absolute value. Note that the `:total` option of `pie` can be used to create pie charts no entirely filled.

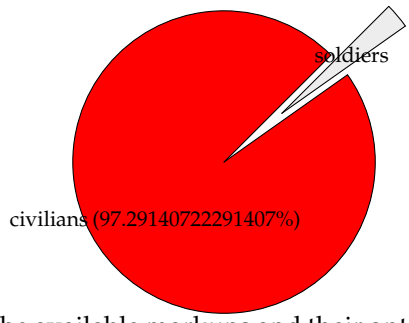
Various options allow the pie layout to be controlled:

Example 40. Specifying the layout of a pie chart

```
;; Another sad pie chart.
;;

(pie :title [Casualties of the Conflict in Iraq since 2003 (source:
English Wikipedia page, 2006-07-23)]
     :ident "pie-iraq-2006"
     :fingers? #f
     :labels 'inside
     :initial-angle 45
     :radius 2

  (slice :weight 100000 :color "red" :detach? #t
         [civilians (,(sliceweight :percentage? #t)%)])
  (slice :weight (+ 2555 229) :color #xeeeeee  [soldiers]))
```

… produces:

Casualties of the Conflict in Iraq since 2003 (source:
English Wikipedia page, 2006-07-23)



The available markups and their options are described below.

```
(pie [:labels 'outside] [:fingers?  #t] [:radius 3] [:total] [:initial-angle
    0] [:title "Pie Chart"] [:class "pie"] [:ident])
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:title html lout latex context info xml

The title of the pie chart.

:initial-angle html lout latex context info xml

The initial angle of the pie, in degrees.

:total html lout latex context info xml

If a number, specifies the "weight" of the whole pie; in this case, if the pie's slices don't add up to that number, then part of the pie is shown as empty. If `#f`, the total that is used is the sum of the weight of each slice.

:radius html lout latex context info xml

The pie's radius. How this value is interpreted depends on the engine used.

:fingers? lout

Indicates whether to add "fingers" (arrows) from labels to slices when labels are outside of slices.

`:labels` html lout latex context info xml

A symbol indicating where slice labels are rendered: `outside` to have them appear outside of the pie, `inside` to have them appear inside the pie, and `legend` to have them appear in a separate legend.

See also `slice`, p. 88 .

(slice [:detach?] [:color "white"] [:weight 1] [:class "pie-slice"] [:ident] label**)**

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:weight` html lout latex context info xml

An integer indicating the weight of this slice.

`:color` html lout latex context info xml

The background color of the slice.

`:detach?` html lout latex context info xml

Indicates whether the slice should be detached from the pie.

`label`

The label of the node. It can contain arbitrary markup, notably instances of `sliceweight`. However, some engines, such as the Ploticus-based rendering, are not able to render markup other than `sliceweight`; consequently, they render the label as though it were markup-free.

See also `pie`, p. 87 , `sliceweight`, p. 88 .

As seen in the examples above, the body of a `slice` markup can contain instances of `sliceweight` to represent the weight of the slice:

(sliceweight [:percentage?] [:class "pie-sliceweight"] [:ident]**)**

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:percentage?` html lout latex context info xml

Indicates whether the slice's weight should be shown as a percentage of the total pie weight or as a raw weight.

See also `slice`, p. 88 .

Chapter 10. Slide Package

This chapter describes the facilities Skribilo offers authoring slides. As usual, in order to access the functionalities described in this chapter, the `(use-modules (skribilo package slide))` expression must be introduced at the beginning of the document.

10.1. Slides and Slide Topics

A `slide` function call creates a slide.

```
(slide :title [:image] [:bg] [:transition] [:vfill] [:vspace] [:number #t]
    [:toc #t] [:class] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:title` html lout latex

The title of the slide.

`:number` html lout latex

The number of the slide (a boolean or an integer).

`:toc` html lout latex

This argument controls if the slide must be referenced in the table of contents and the outline slide that introduces a `slide-topic` (see below).

`:vspace` latex

The boolean `#f` or an integer representing the vertical space size between the title and the body of the slide.

`:vfill` latex

A boolean that is used to control whether a LaTeX `\vfill` is issued at the end of the slide.

`:transition` html latex

The boolean `#f` or a symbol in the list `(split blinds box wipe dissolve glitter)`.

`:bg` html

The background color of the slide.

:image latex

The background image of the slide.

Optionally, one may group slides into *topics* and *subtopics*. Topics and subtopics are just logical grouping under a given title that are meant to provide structure to a set of slides. With their **:outline?** option, these two markups can be used to automatically produce an outline at the beginning of each new (sub)topic, which reminds the audience of the current position within the talk.

```
(slide-topic :title [:class] [:ident] [:toc #t] [:unfold? #t] [:outline?
    #t])
(slide-subtopic :title [:class] [:ident] [:toc #t] [:unfold?]
    [:outline?])
```

:ident html lout latex context info xml

The node identifier.

:class html lout latex context info xml

The node class.

:title html lout latex context info xml

The title of a topic.

:outline? html lout latex context info xml

A boolean (i.e., `#t` or `#f`) telling whether an outline should be produced at the beginning of this topic. The outline will typically list the titles of the different topics, as well as the titles of the slides under the current topic. Note that `slide`s whose **:toc** option is `#f` will not be shown in the outline.

:unfold? lout latex context info xml

If `#t`, then the outline will also show the contents of the current topic.

:toc

This argument controls if the slide must be referenced in the table of contents and the outline slide that introduces a `slide-topic` (see below).

This package understands the following additional customs (see Section 13.1.4):

`slide-outline-title`

The title of outline slides. By default, no title is produced.

`slide-outline-active-color`

The color in which the current slide topic is displayed in outline slides.

`slide-outline-inactive-color`
The color in which inactive slide topics are displayed in outline slides.

10.2. Pause

A `slide-pause` function call introduces a pause in the slide projection. This may not be supported by all engines.

```
(slide-pause)
```

10.3. Slide Vertical Space

The `slide-vspace` introduces a vertical space in the slide.

```
(slide-vspace [:unit 'cm] val)
```

`:unit` lout latex
The unit of the space.

`val`
The size of the vertical space.

10.4. Slide Embed Applications

Embed an application inside a slide.

```
(slide-embed :command [:alt] [:transient-opt] [:transient] [:rgeometry]
    [:geometry] [:geometry-opt "-geometry"] [:arguments '()])
```

`:command` lout latex
The binary file for running the embedded application.

`:arguments` lout
Additional arguments to be passed to the application (a list of strings).

`:geometry-opt` html lout latex context info xml
The name of the geometry option to be sent to the embedded application.

`:geometry` lout latex

The geometry value to be sent.

`:rgeometry` latex

A relative geometry to be sent.

`:transient-opt` latex

The name of the transient option to be sent to the embedded application.

`:transient` latex

The transient value to be sent.

`:alt` html lout latex context info xml

An alternative Skribilo expression to be used if the output format does not support embedded application.

10.5. Example

Here is a complete example of Skribilo slides:

Example 41. Example of Skribilo slides

```
(use-modules (skribilo package slide))

(document :title (color :fg "red" (sf "Skribilo Slides"))
   :author (author :name (it [Bob Smith])
                   :affiliation [The Organization]
                   :address (ref :url "http://example.org/"))

   (slide :title "Table of Contents" :number #f

      ;; "Manually" produce a list of the slides.  This works by traversing
      ;; the whole document tree, looking for `slide' nodes.
      (p
       (resolve (lambda (n e env)
                  (let ((slides (search-down (lambda (n)
                                               (is-markup? n 'slide))
                                             (ast-document n))))
                    (itemize
                     (map (lambda (n)
                            (item (ref :handle (handle n)
                                       :text (markup-option n :text))))
                          slides)))))))

   (slide :title "Introduction" :vspace 0.3

      (p [This is a simple slide, not grouped in any topic.]))

   (slide-topic :title "Interactive Features" :outline? #t

     (slide :title "X11 Clients" :toc #t :vspace 0.3

         (itemize
```

```
            (item "xlock")
            (item "xeyes")
            (item "xterm")))

      (slide :title "Xclock" :toc #t :vspace 0.3

          (center (sf (underline "The Unix xclock client")))
          (slide-vspace 0.3)

          (slide-pause)
          (slide-embed :command "xlock"
                       :alt (frame "Can't run embedded application")))))
```

Chapter 11. Standard Packages

This chapter describes the standard packages that come with Skribilo. Additional packages may be found from the Skribe web page, but they may require slight modifications to work correctly with Skribilo.

In order to use the facilities described in the following sections, the Skribilo source file must contains statements such as:

```
(use-modules (skribilo package THE-PACKAGE))
```

where `THE-PACKAGE` is the desired package. GNU Guile users will recognize that this is Guile's standard way to use Scheme modules. The effect of the `use-modules` clause is to insert the bindings exported by `THE-PACKAGE` into the current module's name space. Note that third-party Skribilo packages can have arbitrary Guile module names. And of course, the `use-modules` clause can be used to import *any* Guile module, not just Skribilo packages.

11.1. Articles

11.1.1. `acmproc`

This package enables producing LaTeX documents conforming to the *ACM proceeding* (ACMPROC) style. It introduces the markup `abstract`:

```
(abstract :postscript [:class "abstract"])
```

`:class` html lout latex context info xml

The node class.

`:postscript` html lout latex context info xml

The URL of the PostScript version of the paper.

11.1.2. `jfp`

This package enables producing LaTeX documents conforming to the *Journal of Functional Programming* (JFP) style. It introduces the markup `abstract`:

```
(abstract :postscript)
```

`:postscript` html lout latex context info xml

The URL of the PostScript version of the paper.

11.1.3. `lncs`

This package enables producing LaTeX documents conforming to the *Lecture Notes on Computer Science* (LNCS) style. It introduces the markups `abstract` and `references`:

```
(abstract :postscript)
```

`:postscript` html lout latex context info xml

The URL of the PostScript version of the paper.

```
(references [:sort])
```

`:sort` html lout latex context info xml

A sort procedure, as for the-bibliography.

See also `the-bibliography`, p. 68 , `bib-sort/authors`, p. 73 .

11.2. Languages

Currently, native language support in Skribilo is rudimentary, limited to the following package. In the future, it should be possible to specify a document's language so that the output engine behaves accordingly, e.g., by choosing the right typographical rules and the right phrases.

11.2.1. `french`

Enables French typesetting and typographical rules.

11.3. `letter`

This package is to be used to authoring simple letters. It redefines the `document` markup.

```
(document :author :date :where [:class "letter"] [:ident])
```

`:ident` html lout latex context info xml

The node identifier.

`:class` html lout latex context info xml

The node class.

`:where` html lout latex context info xml

The location where the letter is posted.

`:date` html lout latex context info xml

The date of the letter.

`:author` html lout latex context info xml

The author of the letter.

11.4. Web

11.4.1. `web-book`

This package provides a convenient mode for rendering books (i.e., documents made of chapters and sections) in HTML. It modifies the `left-margin` custom of the HTML engine (see HTML customs) such that the left margin of each HTML file produced contains a menu allowing for quick navigation within the HTML document.

11.4.2. `web-book2`

This package provides a different way to render books in HTML. Concretely, it prepends a small table of contents to each chapter, section, etc., that appears in an HTML file of its own, making it easy to move around the various HTML files of the document. Compared to `web-book`, it does not clutter the left margin of the HTML pages.

Unlike `web-book`, this package is "purely functional" in that it does not modify the HTML engine customs and writers.

11.4.3. `html-navtabs`

The `html-navtabs` package enables quick navigation inside HTML documents by means of tabs. The produced HTML code uses CSS declarations. The `html-navtabs` package does not introduce any new markups. It is configured via additional engine customs.

11.4.3.1. HTML Engine Customization

`html-navtabs` is to be used in conjunction with the `html-engine` engine. Specifically, it adds the following new customization to this engine:

`html-navtabs` `#<<markup> (it/it22212) 96266592>`

The tabs.

`html-navtabs-padding` `20.0`

Padding above tabs.

`html-navtabs-bar-background` `#f`

Bar background color.

11.4.3.2. Additional Container Options

`html-navtabs` introduces two new containers (i.e., a `document chapter section`, …) attributes: **:html-tabs-bar** and **:no-tabs**. The attribute **:html-tabs-bar** may contain any Skribe expression. It controls the content of the navtabs sub-bar (i.e., a small line above the tabs). The attribute **:no-tabs** disable tabs for this container.

11.4.3.3. Example

Please see the HTML version of the manual for an example.

Chapter 12. Standard Library

This section describes Skribilo's standard library.

12.1. File Functions

The function `include` is particularily useful to spread a long document amongst several files.

(include `file`**)**

`file`

The file containing the nodes to be included. These nodes are included in the document in place of the include call.

See also `*document-path*`, p. 101 .

The given file is searched in the current document path.

Skribilo provides functions to deal with paths. These functions are related to the path that can be specified on the command line, when the Skribilo compiler is invoked (see Chapter 14.)

(*document-path*)

See also `include`, p. 101 , `*image-path*`, p. 41 , `*bib-path*`, p. 66 , `*source-path*`, p. 79 .

`*document-path*` is a procedure as returned by SRFI-39 `make-parameter`. As such, `(*document-path*)` returns the current document path, while `(*document-path* '("." "/some/path"))` changes the value of the current path. This is equivalent to Skribe's `skribe-path` and `skribe-path-set!` functions. The value of `*document-path*` can also be changed using the `-I` command-line option of the compiler (see Chapter 14 for details).

12.2. Configuration Functions

Several functions describing the configuration of Skribilo are exported by the `(skribilo config)` module. First, the `skribilo-version` function returns the version of Skribilo being used as a string.

```
(skribilo-version)
```

For instance, the following piece of code:

```
[This manual documents version ,(bold (skribilo-version)) of Skribi-
lo.]
```

produces the following output

This manual documents version **0.9.3** of Skribilo.

The `skribilo-url` function returns, not surprisingly, the URL of the project:

```
(skribilo-url)
```

The `skribilo-module-directory` returns the directory under which Skribilo modules were installed:

```
(skribilo-module-directory)
```

Similar information can be obtained using the `skribilo-config` program, as described in Section 15.

Chapter 13. Engines

Skribilo documents can be rendered, or output, in a variety of different formats. When using the compiler, which format is used is specified by the `-target` command-line option (see Chapter 14). This command-line option actually specifies the *engine* or "back-end" to be used, which is roughly a set of procedures that translate the input document into the output format. For instance, passing `-target=html` to the compiler instructs it to produce an HTML document using the `html` engine.

This chapter describes procedures allowing the manipulation of engines in Skribilo documents or modules (creation, customization, etc.), as well as the available engines. Currently, the available engines are:

- HTML Engine
- Lout Engine
- LaTeX Engine
- ConTeXt Engine
- Info Engine
- XML Engine

Engine customization provides tight control over the output produced for each particular engine. In particular, it allows the style for each output to be fine-tuned, be it HTML, PDF *via* Lout, or anything else. However, note that such fine-tuning usually requires good knowledge of the output format (e.g., HTML/CSS, Lout, LaTeX).

13.1. Manipulating Engines

13.1.1. Creating Engines

The function `make-engine` creates a brand new engine.

```
(make-engine [:info '()] [:custom '()] [:symbol-table '()] [:delegate]
    [:filter] [:format "raw"] [:version 'unspecified] ident)
```

`:version`

The version number.

`:format`
: The output format (a string) of this engine.

`:filter`
: A string filter (a function).

`:delegate`
: A delegate engine.

`:symbol-table`
: The engine symbol table.

`:custom`
: The engine custom list.

`:info`
: Miscellaneous.

`ident`
: The name (a symbol) of the new engine.

The function `copy-engine` duplicates an existing engine.

(copy-engine [:custom] [:symbol-table] [:delegate] [:filter] [:version 'unspecified] ident e**)**

`:version`
: The version number.

`:filter`
: A string filter (a function).

`:delegate`
: A delegate engine.

`:symbol-table`
: The engine symbol table.

`:custom`
: The engine custom list.

`ident`
: The name (a symbol) of the new engine.

`e`
: The old engine to be duplicated.

13.1.2. Retrieving Engines

The `find-engine` function searches in the list of defined engines. It returns an `engine` object on success and `#f` on failure.

```
(find-engine [:version 'unspecified] id)
```

`:version`
: An optional version number for the searched engine.

`id`
: The name (a symbol) of the engine to be searched.

13.1.3. Engine Accessors

The predicate `engine?` returns `#t` if its argument is an engine. Otherwise, it returns `#f`. In other words, `engine?` returns `#t` for objects created by `make-engine`, `copy-engine`, and `find-engine`.

```
(engine? obj)
```

`obj`
: The checked object.

The following functions return information about engines.

```
(engine-ident obj)
(engine-format obj)
(engine-customs obj)
(engine-filter obj)
(engine-symbol-table obj)
```

`obj`
: The engine.

13.1.4. Engine Customs

Engine customs are locations where dynamic informations relative to engines can be stored. Engine custom can be seen a global variables that are specific to engines. The function

`engine-custom` returns the value of a custom or `#f` if that custom is not defined. The function `engine-custom-set!` defines or sets a new value for a custom.

(engine-custom `e id`**)**

`e`
: The engine (as returned by `find-engine`).

`id`
: The name of the custom.

(engine-custom-set! `e id val`**)**

`e`
: The engine (as returned by `find-engine`).

`id`
: The name of the custom.

`val`
: The new value of the custom.

In the documentation of available engines that follows, a list of available customs is shown for each engine, along with each custom's default value and a description.

13.1.5. Writing New Engines

Writing new engines (i.e., output formats) and making them available to Skribilo is an easy task. Essentially, this boils down to instantiating an engine using `make-engine` and registering *markup writers* using the `markup-writer` procedure for all supported markups (e.g., `chapter`, `bold`, etc.)[1].

Most likely, you will want to make your new engine visible so that `find-engine` and consequently the `-target` command-line option can find it. To that end, a few rules must be followed:

1. your engine must be enclosed in a Guile Scheme module under the `skribilo engine` hierarchy; for instance, if the engine is named `foo`, then it should be in a module called `(skribilo engine foo)`;

2. the engine itself as returned by `make-engine` must be bound, in that module, to a variable called, say, `foo-engine`;

[1] FIXME: Markup writers are not currently documented, but looking at the source of an engine will give you the idea, trust me.

3. finally, the `(skribilo engine foo)` module must be in Guile's load path; for instance, you can adjust the `GUILE_LOAD_PATH` environment variable.

This is all it takes to extend Skribilo's set of engines! Note that this mechanism is the same as that of *readers* (see Section 2.5).

13.2. HTML Engine

The HTML engine produces—guess what!—HTML output. It can be customized in various ways, as shown below.

13.2.1. HTML Customization

`favicon` `#f`

The name of an image file of the URL image. The `favicon` custom can be either bound to a string which is the name of the image, or to a procedure of two arguments, a node and an engine that returns the file name of the icon. This can be used to use different icons per chapter or section.

`charset` `"ISO-8859-1"`

The character set used for the document.

`javascript` `#f`

Enable/disable Javascript support.

`head` `#f`

A string included in the HTML header.

`css` `()`

The URL or a list of URLs of CSS used by the document.

`inline-css` `()`

The file or a list of files inlined inside the header's style section. The custom `inline-css` should be used in replacement of the `css` custom in order to produce stand alone HTML documents.

`js` `()`

A URL or a list of URLs of JavaScript programs used by the document.

`emit-sui` `#f`

Emit a SUI file for this document (see Section 4.4 for details).

`background` `#f`

The background color of the document.

`foreground` `#f`

The foreground color of the document.

`margin-padding` `3`

Margins padding.

`left-margin` `#f`

A procedure of two arguments producing the left margin of the document.

`chapter-left-margin` `#f`

A procedure of two arguments producing the left margin of the document.

`section-left-margin` `#f`

A procedure of two arguments producing the left margin of the document.

`left-margin-font` `#f`

The font of the left margin.

`left-margin-size` `17.0`

The width of the left margin.

`left-margin-background` `#f`

The background color of the left margin.

`left-margin-foreground` `#f`

The foreground color of the left margin.

`right-margin` `#f`

A procedure of two arguments producing the right margin of the document.

`chapter-right-margin` `#f`

A procedure of two arguments producing the right margin of the document.

`section-right-margin` `#f`

A procedure of two arguments producing the right margin of the document.

`right-margin-font` `#f`

The font of the right margin.

`right-margin-size` `17.0`

The width of the right margin.

`right-margin-background` `#f`

The background color of the right margin.

`right-margin-foreground` `#f`

The foreground color of the right margin.

`author-font` `#f`

The author font.

`title-font` `#f`

The title font.

`title-background` `#f`

The title background color.

`title-foreground` `#f`

The title foreground color.

`file-title-separator` `(unquote (! " &#8212; "))`

A text to be inserted in between the document title and the chapter or section title when the chapter or section is rendered in a separate file.

`file-name-proc` `(unquote html-file-default)`

A two-argument procedure that should return a string. This procedure is to be passed a node and an engine and should return a file name for the HTML page corresponding to this node.

`index-header-font-size` `#f`

The index header font size.

`chapter-number->string` `number->string`

A procedure of one argument for rendering chapter numbers.

`chapter-file` `#f`

A boolean specifying if chapters are rendered in separate HTML file (see `chapter` markup).

`section-title-start` `"<h3>"`

The HTML sequence for starting section title.

`section-title-stop` `"</h3>"`

The HTML sequence for stopping section title.

`section-title-background` `#f`

The background color of section title.

`section-title-foreground` `#f`

The foreground color of section title.

`section-title-number-separator` `" "`

The section title number separator.

`section-number->string` `number->string`

A procedure of one argument for rendering section numbers.

`section-file` `#f`

A boolean specifying if sections are rendered in separate HTML file (see `section` markup).

`subsection-title-start` `"<h3>"`

The HTML sequence for starting subsection title.

`subsection-title-stop` `"</h3>"`

The HTML sequence for stopping subsection title.

`subsection-title-background` `#f`

The background color of subsection title.

`subsection-title-foreground` `#f`

The foreground color of subsection title.

`subsection-title-number-separator` `" "`

The subsection title number separator.

`subsection-number->string` `number->string`

A procedure of one argument for rendering subsection numbers.

`subsection-file` `#f`

A boolean specifying if subsections are rendered in separate HTML file (see `subsection` markup).

`subsubsection-title-start` `"<h4>"`

The HTML sequence for starting subsubsection title.

`subsubsection-title-stop` `"</h4>"`

The HTML sequence for stopping subsubsection title.

`subsubsection-title-background` `#f`

The background color of subsubsection title.

`subsubsection-title-foreground` `#f`

The foreground color of subsubsection title.

`subsubsection-title-number-separator` `" "`

The subsubsection title number separator.

`subsubsection-number->string` `number->string`

A procedure of one argument for rendering subsubsection numbers.

`subsubsection-file` `#f`

A boolean specifying if subsubsections are rendered in separate HTML file (see `subsubsection` markup).

`source-color` `#t`

A boolean enabling/disabling color of source code (see `source` markup).

`source-comment-color` `"#ffa600"`

The source comment color.

`source-error-color` `"red"`

The source error color.

`source-define-color` `"#6959cf"`

The source define color.

`source-module-color` `"#1919af"`

The source module color.

`source-markup-color` `"#1919af"`

The source markup color.

`source-thread-color` `"#ad4386"`

The source thread color.

`source-string-color` `"red"`

The source string color.

`source-bracket-color` `"red"`

The source bracket color.

`source-type-color` `"#00cf00"`

The source type color.

`image-format` `("png" "gif" "jpg" "jpeg")`

The image formats for this engine.

13.3. Lout Engine

The Lout engine produces documents for the Lout typesetting system, which is then suitable for the production of PostScript/PDF files for printing. Lout is a typesetting system comparable to TeX/LaTeX in functionality. However, it is based on a lazy, purely functional programming language and makes it easy to customize document layout; it is also lightweight compared to typical LaTeX installations, consuming less than 10 MiB of disk space.

Skribilo's Lout engine provides lots of customization opportunities (currently more than the LaTeX engine), which are shown below. It also enhances Lout by adding new features: PDF bookmarks, high-level interface to the use of dropped capitals, improved paragraph indentation, etc.

13.3.1. Lout Customization

`document-type` `doc`

A symbol denoting the underlying Lout document type, i.e., one of `doc` (the default), `report`, `book` or `slides`. Note that these document types are not interchangeable: `slides` should be used only when using the `slides` package; `report` and `book` do not permit text in the body of a document outside chapters. Also, these document types provide different layout features, `book` being the "richest" one; in addition, some of the customs below do not apply to all these document types.

`document-include` `auto`

Document style file include line (a string such as `@Include { my-doc-style.lout }`) or the symbol `auto` in which case the include file is deduced from `document-type`.

`includes` `"@SysInclude { tbl }\n"`

A string containing `@Include` directives.

`inline-definitions-proc` `(unquote lout-definitions)`

A procedure that is passed the engine and returns Lout definitions to be included at the beginning of the document as a string.

`encoding` `"ISO-8859-1"`

The encoding of the output document[1]. As of Lout 3.39, only `"ISO-8859-1"` and `"ISO-8859-2"` are supported.

`initial-font` `"Palatino Base 10p"`

Lout specification of the document font.

```
initial-break    (unquote      (string-append      "unbreakablefirst      "
                "unbreakablelast " "hyphen adjust 1.2fx"))
```

Lout specification of the default paragraph breaking style.

`initial-language` `"English"`

Lout specification of the document's language. This is used to select hyphenation rules, among other things.

`column-number` `1`

Number of columns.

`first-page-number` `1`

Number of the first page.

`page-orientation` `portrait`

A symbol denoting the page orientation, one of `portrait`, `landscape`, `reverse-portrait` or `reverse-landscape`.

`cover-sheet?` `#t`

For `report`, this boolean determines whether a cover sheet should be produced. The `doc-cover-sheet-proc` custom may also honor this custom for `doc` documents.

`date-line` `#t`

---

[1]This option is supported when Guile 2.0+ is being used.

For `report` and `slide`, determines whether a date line will appear on the first page (if it's a boolean), or what date line will appear (if it's not a boolean).

`abstract` `#f`

For `report`, this can be an arbitrary Skribe expression for use as an abstract.

`abstract-title` `#t`

For `report`, the title/name of the abstract. If `#f` then no abstract title is produced. If `#t`, then a default abstract title is chosen according to `initial-language`.

`publisher` `#f`

For `book`, the publisher.

`edition` `#f`

For `book`, the edition.

`before-title-page` `#f`

For `book`, an expression that will appear before the title page.

`on-title-page` `#f`

For `book`, the expression used as the title page.

`after-title-page` `#f`

For `book`, an expression that will appear right after the title page.

`at-end` `#f`

For `book`, an expression that will appear at the end of the book, on a page of its own.

`optimize-pages?` `#f`

A boolean indicating whether to optimize pages. Refer to Lout's User's Guide for caveat.

`doc-cover-sheet-proc` `(unquote lout-make-doc-cover-sheet)`

For `doc`, a procedure that produces the title or cover sheet. When invoked, the procedure is passed the `document` node and the engine.

`bib-refs-sort-proc` `#f`

Kept for backward compability, do not use.

`paragraph-gap` `"\n//1.0vx @ParaIndent @Wide &{0i}\n"`

Lout code for paragraph gaps. Note that the default value is not `@PP` as one would expect but is instead similar to `@PP` with `@ParaGap` equal to `1.0vx`, which means that

a regular inter-line space is used as inter-paragraph space. This differs from Lout's default where the inter-paragraph space is larger than the inter-line space, but looks better, at least in the author's eyes.

`first-paragraph-gap` `"\n@LP\n"`

Gap for the first paragraph within a container (e.g., the first paragraph of a chapter). This allows paragraphs to have a different indentation depending on whether they are the first paragraph of a section or not. By default, the first paragraph is not indented and subsequent paragraph are indented.

`drop-capital?` `#f`

A boolean or predicate indicating whether drop capitals should be used at the beginning of paragraphs. When invoked, the predicate is passed the node at hand and the engine.

`drop-capital-lines` `2`

Number of lines over which dropped capitals span. Only 2 and 3 are currently supported.

`use-header-rows?` `#f`

For multi-page `table`s, setting this to `#t` allows header rows to be repeated on each new page. However, it appears to be buggy at the moment.

`use-lout-footnote-numbers?` `#f`

Tells whether to use Lout's footnote numbering scheme or Skribilo's number. Using Lout's numbering scheme may yield footnote numbers that are different from those obtained with other engines, which can be undesirable.

`transform-url-ref-proc` `(unquote lout-split-external-link)`

A procedure that takes a URL `ref` markup and returns a list containing (maybe) one such `ref` markup. This custom can be used to modify the way URLs are rendered. The default value is a procedure that limits the size of the text passed to Lout's `@ExternalLink` symbols to work around the fact that `@ExternalLink` objects are unbreakable. In order to completely disable use of `@ExternalLink`, just set it to `markup-body`.

`toc-leader` `"."`

A string, which is the leader used in table-of-content entries.

`toc-leader-space` `"2.5s"`

Inter-leader space in table-of-contents entries.

`toc-entry-proc` `(unquote lout-make-toc-entry)`

Procedure that takes a large-scale structure (chapter, section, etc.) and the engine and produces the number and possibly title of this structure for use in table-of-contents.

`lout-program-name` `"lout"`

The `lout` program path, only useful when producing `lout-illustration` on other engines.

`lout-program-arguments` `()`

List of additional arguments that should be passed to `lout`, e.g., `("-I foo" "-I bar")`.

`make-pdf-docinfo?` `#t`

Tells whether to produce PDF "docinfo", i.e., meta-information with title, author, etc.

`pdf-title` `#t`

Title for use as the PDF document meta-information. If `#t`, the `document`'s **:title** is used.

`pdf-author` `#t`

Author for use as the PDF document meta-information. If `#t`, the `document`'s **:author** is used.

`pdf-keywords` `#f`

Keywords (a list of string) in the PDF document information. This custom is deprecated, use the **:keywords** option of `document` instead.

`pdf-extra-info` `(("SkribiloVersion" (unquote (skribilo-version))))`

A list of key-value pairs (strings) to appear in the PDF meta-information.

`make-pdf-outline?` `#t`

Tells whether to produce a PDF outline (aka. "bookmarks").

`pdf-bookmark-title-proc` `(unquote lout-pdf-bookmark-title)`

Procedure that takes a node and an engine and return a string representing the title of that node's PDF bookmark.

`pdf-bookmark-node-pred` `(unquote lout-pdf-bookmark-node?)`

Predicate that takes a node and an engine and returns true if that node should have a PDF outline entry.

`pdf-bookmark-closed-pred` `(unquote (lambda (n e) (not (and (markup? n) (memq (markup-markup n) (quote (chapter slide slide-topic)))))))`

Predicate that takes a node and an engine and returns true if the bookmark for that node should be closed ("folded") when the user opens the PDF document.

`color?` `#t`

Indicate whether to use colors or not.

`source-color` `#t`

A boolean enabling/disabling color of source code (see `source` markup).

`source-comment-color` `"#ffa600"`

The source comment color.

`source-define-color` `"#6959cf"`

The source define color.

`source-module-color` `"#1919af"`

The source module color.

`source-markup-color` `"#1919af"`

The source markup color.

`source-thread-color` `"#ad4386"`

The source thread color.

`source-string-color` `"red"`

The source string color.

`source-bracket-color` `"red"`

The source bracket color.

`source-type-color` `"#00cf00"`

The source type color.

13.3.2. Additional Markup

The `(skribilo engine lout)` module also exports a new markup called `lout-illustration`, which provides an engine-independent way to include illustrations written in Lout,

such as `@Diag` pictures. When an engine other than Lout is used, `lout-illustration` are first automatically translated to EPS (using Lout's `@Illustration`) and then to whatever image format is supported by the engine (see Section 3.12).

(lout-illustration :alt :ident [:file] illustration…)

:ident html lout latex context info xml

An identifier. This identifier is also used as the basis of the EPS file name with non-Lout engines.

:file

If different from `#f`, this specifies a file where the Lout illustration is stored.

:alt

A string displayed on display devices not capable of displaying images, as for `image`.

`illustration…`

The illustration itself if **:file** is `#f`.

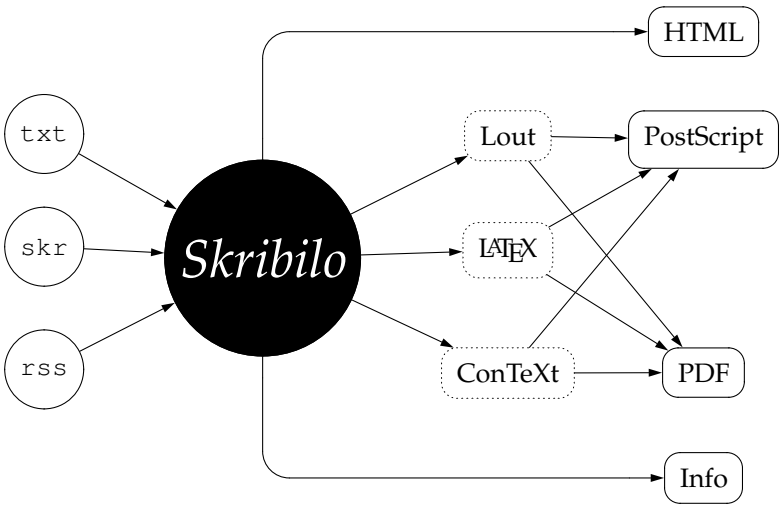
The following example shows a simple diagram. When using the `lout` engine, the diagram is integrated in-line in the document. When using other engines, it is integrated using `image`.

Example 42. A Lout illustration

```
(use-modules (skribilo engine lout))

(lout-illustration :ident "document-toolchain"
                   :alt "a document toolchain" "

# This is Lout code to produce a diagram.
@Diag
  aoutline { circle }
  afont { Courier Base 1f }
  boutline { circle }
  bfont { Palatino Slope 2f }
  bpaint { black }
  bformat { white @Color @Body }
  coutline { curvebox }
  coutlinestyle { dotted }
  doutline { curvebox }
{
  @Tbl
    strut { yes }
    indent { ctr }
    aformat { @Cell A | @Cell marginhorizontal { 2.0fe } B |
              @Cell C | @Cell D }
    amargin { 1.0fe }
  {
    @Rowa D { W:: @DNode HTML }
    @Rowa A { A:: @ANode txt } C { P:: @CNode Lout }
```

```
            D { X:: @DNode PostScript }
      @Rowa A { B:: @ANode skr } B { S:: @BNode Skribilo }
            C { Q:: @CNode @LaTeX }
      @Rowa A { C:: @ANode rss } C { R:: @CNode ConTeXt }
            D { Y:: @DNode PDF }
      @Rowa D { Z:: @DNode Info }
    }

    //

    # input arrows
    @Arrow from { A } to { S }
    @Arrow from { B } to { S }
    @Arrow from { C } to { S }

    # arrows to intermediate files
    @Arrow from { S } to { P }
    @Arrow from { S } to { Q }
    @Arrow from { S } to { R }

    # PS/PDF incoming arrows
    @Arrow from { P } to { X }
    @Arrow from { P } to { Y }
    @Arrow from { Q } to { X }
    @Arrow from { Q } to { Y }
    @Arrow from { R } to { X }
    @Arrow from { R } to { Y }

    # HTML and Info
    @Link from { S } to { W } arrow { yes } path { vhcurve }
    @Link from { S } to { Z } arrow { yes } path { vhcurve }

  }
  ")
```

… produces:



13.4. LaTeX Engine

Not surprisingly, the LaTeX engine produces LaTeX output, which can then be used to produce high-quality PostScript of PDF files for printing.

13.4.1. LaTeX Customization

`documentclass` `"\\documentclass{article}"`

A string declaring the LaTeX document class.

`encoding` `"UTF-8"`

The encoding of the output document[1].

`class-has-chapters?` `#f`

A boolean indicating whether the document class has a `chapter` markup. If `#f`, then Skribilo's `chapter` is mapped to LaTeX' `section`, and so on.

`usepackage` `"\\usepackage{epsfig}\n"`

The boolean `#f` if no package is used or a string declaring The LaTeX packages.

`predocument` `"\\newdimen\\oldframetabcolsep\n\\newdimen\\oldcolortabcolsep\n\\newdimen\\oldpretabcolsep\n"`

The boolean `#f` or a string to be written before the \begin{document} statement.

`postdocument` `#f`

The boolean `#f` or a string to be written after the \begin{document} statement.

`maketitle` `"\\date{}\n\\maketitle"`

The boolean `#f` or a string to be written after the \begin{document} statement for emitting the document title.

`color` `#t`

Enable/disable colors.

`%font-size` `0`

`source-color` `#t`

A boolean enabling/disabling color of source code (see `source` markup).

[1]This option is supported when Guile 2.0+ is being used.

`source-comment-color` `"#ffa600"`

The source comment color.

`source-error-color` `"red"`

The source error color.

`source-define-color` `"#6959cf"`

The source define color.

`source-module-color` `"#1919af"`

The source module color.

`source-markup-color` `"#1919af"`

The source markup color.

`source-thread-color` `"#ad4386"`

The source thread color.

`source-string-color` `"red"`

The source string color.

`source-bracket-color` `"red"`

The source bracket color.

`source-type-color` `"#00cf00"`

The source type color.

`color-usepackage` `"\\usepackage{color}\n"`

The LaTeX package for coloring.

`hyperref` `#t`

Enables/disables hypererrf.

`hyperref-usepackage` `"\\usepackage[setpagesize=false]{hyperref}\n"`

The LaTeX package for hyperref.

`image-format` `("eps")`

The image formats for this engine.

`index-page-ref` `#t`

Indexes use page references.

13.4.2. LaTeX Document Class

The default setting of the Skribilo LaTeX engine is to produce a document using the `article` document class. In order to produce a document that uses a document class defining the `chapter` command (unlike the `article` class), the engine must be customized. Changing this setting can be done with expressions such as:

```
(let ((le (find-engine 'latex)))
   (engine-custom-set! le 'documentclass "\\documentclass{book}")
   (engine-custom-set! le 'class-has-chapters? #t))
```

13.5. ConTeXt Engine

The `context` engine produces documents for the ConTeXt document layout system, which can then be used to produce high-quality PostScript or PDF output.

13.5.6. ConTeXt Customization

`document-style` `"book"`

A string describing the document style.

`user-style` `#f`

A string denoting a the name of a file specifying user customization

`font-type` `"roman"`

A string denoting the default font family.

`font-size` `11`

An integer representing the default font size.

`image-format` `("jpg")`

A list of supported image formats.

`source-comment-color` `"#ffa600"`

The source comment color.

`source-error-color` `"red"`

The source error color.

`source-define-color` `"#6959cf"`

The source define color.

`source-module-color` `"#1919af"`

The source module color.

`source-markup-color` `"#1919af"`

The source markup color.

`source-thread-color` `"#ad4386"`

The source thread color.

`source-string-color` `"red"`

The source string color.

`source-bracket-color` `"red"`

The source bracket color.

`source-type-color` `"#00cf00"`

The source type color.

13.7. Info Engine

The `info` engine produces GNU Info files for on-line browsing with GNU Emacs or with the stand-alone Info reader of GNU Texinfo.

For each `chapter`, `section`, etc., an Info node is created, whose name is inferred from the **:title** option. However, Info node names have to be unique, which the **:title** options are not necessarily. Thus, the Info engine does two things:

1. It warns you about duplicate Info node titles.
2. It allows you to choose a different node name to avoid conflicts, using the **:info-node** option of `chapter`, etc.

Most markups shown in Chapter 3 are meaningfully rendered in Info, including tables. The `image` markup is also implemented: the Info reader in Emacs 23 and later is able to display them, or to display the alternate text (the body of the `image` markup) when running in text mode.

13.8. XML Engine

The XML engine produces a simple XML representation of Skribilo documents that is essentially a one-to-one mapping of the input document. For instance, `chapter` markups are turned into `<chapter>` nodes, etc.

13.8.1. XML Customization

Chapter 14. Skribilo Compiler

This chapter introduces the Skribilo compiler, i.e., the tool that turns input documents into various output formats.

Synopsis

skribilo [*options*] [*input*]…

Description

The `skribilo` compiler turns Skribilo input documents into one of a variety of output formats, including HTML, LaTeX and Lout. The input format is specified using the `-reader` command-line option, while the output format is specified using the `-target` option. These options and others are described below.

Suffixes

A number of file name extensions are used by convention:

`.skb`
: a Skribilo or Skribe source file.

`.html`
: an HTML target file.

`.lout`
: a Lout target file.

`.tex`
: a TeX, LaTeX or ConTeXt target file.

`.sui`
: a Skribe URL index file.

Options

The options supported by the `skribilo` compiler are listed below. They follow the usual GNU convention, i.e., each option can have both a short name (a hyphen followed by a single character) and a long name (two hyphens followed by a name).

`-h, -help`
: Produce a help message.

`-V, -version`
: Show program version.

`-R, -reader=reader`
: Use `reader` to read the input file, i.e., as the format of the input file. Currently, two formats are supported: `skribe`, which corresponds to the Skribe syntax (see Chapter 2), or `outline`, which corresponds to plain text (markup-less) following the structuring conventions of Emacs' Outline mode (see Section 2.2).

`-t, -target=engine`
: Use `engine` as the engine, i.e., as the output format. For details on engines and for a list of supported engines, see Chapter 13.

`-c, -custom=custom=value`
: Set engine custom `custom` to `value`, a constant. See Section 13.1.4 for more information on customs.

`-o, -output=file`
: Write output to `file`.

`-compat=compat`
: Use `compat` as the compatibility mode. This defaults to `skribilo`. Specifying `skribe` enables the Skribe compatibility mode, making it possible to compile most Skribe documents. Technically, the `skribe` compatibility mode populates the name space of Skribilo documents with bindings available to Skribe documents and that are not available by default to Skribilo documents[1] (e.g., SRFI-1 functions, Bigloo's hash table API, etc.); for Skribe functions not available in Skribilo, such as `skribe-load`, a compatible implementation is provided.

`-I, -doc-path=dir`
: Prepend `dir` to the document include path.

`-B, -bib-path=dir`
: Prepend `dir` to the bibliography include path.

`-S, -source-path=dir`
: Prepend `dir` to the source include path.

---

[1] Skribe uses a single name space for all the code of packages, documents, in addition to bindings provided by the underlying Scheme implementation.

`-P, -image-path=dir`
: Prepend `dir` to the image include path.

`-U, -sui-path=dir`
: Prepend `dir` to the Skribe URL Index (SUI) search path (see Section 4.4 for details).

`-b, -base=base`
: Strip `base` (an arbitrary string, typically an URL) from all hyperlinks when producing HTML files.

`-e, -eval=expr`
: Prepend `expr` to the list of expressions to be evaluated before the input document is processed. `expr` is evaluated in the document's environment/module; thus, this option can be used to pass parameters to the document, e.g., with `-e '(define chbouib-enabled? "yes")'`.

`-p, -preload=file`
: Pre-load `file` before processing the input document. `file` is evaluated in the document's name space and should be a regular Scheme file, i.e., it cannot use the Skribe syntax.

`-v, -verbose[=level]`
: Be verbose, unless `level` is `0`.

`-w, -warning[=level]`
: Issue warnings, unless `level` is `0`.

`-g, -debug[=arg]`
: Issue debugging output, unless `arg` is `0`. If `arg` is not a number, it is interpreted as a symbol to be watched.

`-no-color`
: By default, debugging output is colored on capable terminals such as `xterm` or the Linux console (check your `TERM` environment variable). This option turns coloring off.

Environment Variables

The `skribilo` command does not pay attention to any specific environment variable. In particular, it does not honor the `SKRIBEPATH` variable that is recognized by Skribe. Instead, you should use the `-I` command-line option to specify the load path of *documents* (see `include`), or, alternatively, change the value of the `GUILE_LOAD_PATH` variable, which affects Guile's own module load path.

Chapter 15. Getting Configuration Information

This chapter presents `skribilo-config`, a stand-alone program that gives information about the current configuration.

Synopsis

skribilo-config [*options*]…

Description

The `skribilo-config` program gives information about the Skribilo configuration, such as the module installation path, version number, etc. The full list of supported options and their meaning is the following:

```
Usage: skribilo-config [OPTIONS]
Display the configuration of Skribilo.

  -version, -v               Show Skribilo version.
  -help, -h                Show a list of options.

  -prefix, -p                Prefix that was given during the build
  -module-dir, -m        Display the Guile module directory
  -doc-dir, -d               Display the documentation directory lo-
cation
  -emacs-dir, -e        Display the emacs directory location
  -scheme, -s               Display the configured Scheme implementation

Report bugs to <skribilo-users@nongnu.org>.
```

Note that the same information can be obtained through the programming interface exposed by the `(skribilo config)` module (see Section 12.2).

Chapter 16. Editing Skribilo Programs

Skribilo documents can be typed in. The `outline` syntax (see Section 2.2) can be easily typed in with any editor, although the Outline and Org modes found in GNU Emacs and XEmacs make it more convenient to deal with this format. For instance, they allow section contents to be hidden, leaving only the section headings visible; Org Mode can also highlight forms that denote emphasis, and provide proper display of Org-Mode-style hyperlinks (well, not surprisingly).

When using the Skribe syntax (see Section 2.1), it is highly recommended to use GNU Emacs or XEmacs. In addition to parentheses matching, these editors provide syntax highlighting (or "fontification") through the Skribe Mode described below.

16.1. Skribe Emacs Mode

The Skribilo distribution contains a minor mode dedicated to Skribe edition originally written by Manuel Serrano. This mode provides *fontification* and indentation of Skribe programs. In this manual, we present the two most important key bindings specific to this mode:

- tab indents the current line.
- M-C-q indents a whole Skribe expression.

To use the Skribe/Skribilo Emacs mode, you need to tell Emacs that when the Emacs Lisp `skribe-mode` function is needed it has to be loaded from the `skribe.el` file:

```
(autoload 'skribe-mode "skribe.el" "Skribe mode." t)
```

The `skribe.el` file must in the path described by the Emacs Lisp `load-path` variable.

The `skribe` mode is a minor mode. It is intended to be used with a Lisp or Scheme mode. Hence, to use the `skribe` mode you will have to use the following Emacs commands:

```
M-x scheme-mode
M-x skribe-mode
```

Chapter 17. List of examples

Index