Simple Tree Text Markup: Simple Markup for Display as Text or in GUI

Version 9.2.0.6

Mike Sperber

July 4, 2026

This is a tree-based combinator library for simple markup, mainly for displaying messages in a REPL. It features horizontal and vertical composition as well as framed markup. Its main distinguishing feature is its ability to embed source locations, which can be rendered as links.

This package comes with separate modules for *inspecting* and *constructing* markup - `simple-tree-text-markup/data` and `simple-tree-text-markup/construct`, respectively. Markup can also be constructed through a custom output port, supplied by `simple-tree-text-markup/port`.

There's also a module `simple-tree-text-markup/text` that renders markup to text. Rendering markup to GUI is quite context-specific. Hence, the code for rendering to GUIs is implemented with specific applications, such as DrRacket or the test engine.

1 Markup Representation

```
(require simple-tree-text-markup/data)
      package: simple-tree-text-markup-lib
```

This module defines the representation for markup as a set of struct definitions. It should be required when inspecting markup, For constructing markup, see `simple-tree-text-markup/construct`.

A markup object can be one of the following:

- a string
- an `empty-markup`
- a `horizontal-markup`
- a `vertical-markup`
- a `srcloc-markup`
- a `framed-markup`
- an `image-markup`
- a `number-markup`

```
(markup? object) → boolean?
  object : any/c
```

Returns `#t` if *object* is a markup object, `#f` otherwise.

```
(struct empty-markup ())
```

This is an empty markup object, which consumes no space.

```
(struct horizontal-markup (markups))
  markups : (listof markup?)
```

This markup object contains several sub-markups, which will be arranged horizontally when rendered.

```
(struct vertical-markup (markups))
  markups : (listof markup?)
```

This markup object contains several sub-markups, which will be arranged vertically when rendered.

```
(struct srcloc-markup (srcloc markup))
  srcloc : srcloc?
  markup : markup?
```

This markup object represents a link to a source location, represented by `srcloc`, where the link visualization is represented by `markup`.

```
(struct framed-markup (markup))
  markup : markup?
```

This markup object puts a frame around `markup`.

```
(struct image-markup (data alt-markup))
  data : any/c
  alt-markup : markup?
```

This markup object represents an image. The `data` contains the image data. The format is not exactly specified, but a graphical renderer should accept `bitmap%`, `snip%`, and `record-dc-datum` objects.

If rendering of `data` is not possible, `alt-markup` can be substituted.

```
(struct record-dc-datum (datum width height))
  datum : any/c
  width : natural-number/c
  height : natural-number/c
```

This represents an image, containing the result the `get-recorded-datum` from `record-dc%`, as well as the width and height of that image.

```
(struct number-markup (number
                       exact-prefix
                       inexact-prefix
                       fraction-view))
  number : number?
  exact-prefix : (or/c 'always 'never 'when-necessary)
  inexact-prefix : (or/c 'always 'never 'when-necessary)
  fraction-view : (or/c 'mixed 'improper 'decimal #f)
```

This represents a number to be rendered in a format that can be read back.

The `exact-prefix` argument specifies whether the representation should carry a `#e` prefix: Always, never, or when necessary to identify a representation that would otherwise be considered inexact.

Similarly for `inexact-prefix`. Note however that `'when-necessary` is usually equivalent to `'never`, as inexact numbers are always printed with a decimal dot, which is sufficient to identify a number representation as inexact.

The `fraction-view` field specifies how exact non-integer reals - fractions - should be rendered: As a mixed fraction, an improper fraction, or a decimal, possibly identifying periodic digits. For `'decimal`, if it's not possible to render the number as a decimal exactly, a fraction representation might be generated. For `'mixed` an improper fraction representation might be generated if a mixed representation could not be read back.

If `fraction-view` is `#f`, this option comes from some unspecified user preference.

```
(markup-folder combine identity extractors)
 → (markup? . -> . any/c)
  combine : procedure?
  identity : any/c
  extractors : (listof pair?)
```

This creates a procedure that folds over a markup tree using a monoid: That procedure maps every node of the markup tree to an element of the monoid, and returns the result of combining those values.

The monoid itself is defined by *combine* (its binary operation) and *identity* (its identity / neutral element).

The *extractors* list consists of pairs: Each pair consists of a predicate on markup nodes (usually `string?`, `empty-markup?` etc.) and a procedure to map a node, for which the predicate returns a true value, to an element of the monoid.

The following example extracts a list of source locations from a markup tree:

```
(define markup-srclocs
  (markup-folder append '()
                 `((,srcloc-markup? . ,(lambda (markup)
                                         (list (srcloc-markup-
srcloc markup)))))))
```

```
(transform-markup mappers markup) → markup?
  mappers : (listof pair?)
  markup : markup?
```

This procedure transforms markup by replacing nodes. The *mappers* argument is a list of pairs. Each pair consists of a predicate on markup nodes (usually `string?`, `empty-markup?` etc.) and a procedure that accepts as argument the struct components of the corresponding node, where the markup components have been recursively passed through `transform-markup`. The node is replaced by the return value of the procedure.

The following example transforms each piece of image data in a markup tree:

```
(define (markup-transform-image-data transform-image-data markup)
  (transform-markup
   `((,image-markup? . ,(lambda (data alt-markup)
                          (image-markup (transform-image-
data data) alt-markup))))
   markup))
```

2 Markup Construction

```
(require simple-tree-text-markup/construct)
          package: simple-tree-text-markup-lib
```

While the struct definitions in `simple-tree-text-markup/data` can also be used for constructing markup, the procedures exported here are somewhat more convenient to use, and do a fair amount of normalization upon constructions.

```
(srcloc-markup srcloc markup) → markup?
  srcloc : srcloc?
  markup : markup?
```

This constructs a markup object that will represent a link to a source location, represented by *srcloc*, where the link visualization is represented by *markup*.

```
(framed-markup markup) → markup?
  markup : markup?
```

This markup constructor puts a frame around *markup*.

```
empty-markup : markup?
```

This is the empty markup object.

```
empty-line : markup?
```

This is a markup object representing an empty line, i.e. empty vertical space.

```
(number  number
        [#:exact-prefix exact-prefix
         #:inexact-prefix inexact-prefix
         #:fraction-view fraction-view]) → markup?
  number : number?
  exact-prefix : (or/c 'always 'never 'when-necessary) = 'never
  inexact-prefix : (or/c 'always 'never 'when-necessary)
                 = 'never
  fraction-view : (or/c #f 'mixed 'improper 'decimal) = #f
```

This constructs markup for a number to be rendered in a format that can be read back.

The *exact-prefix* argument specifies whether the representation should carry a `#e` prefix: Always, never, or when necessary to identify a representation that would otherwise be considered inexact.

Similarly for *inexact-prefix*. Note however that 'when-necessary is usually equivalent to 'never, as inexact numbers are always printed with a decimal dot, which is sufficient to identify a number representation as inexact.

The *fraction-view* field specifies how exact non-integer reals - fractions - should be rendered: As a mixed fraction, an improper fraction, or a decimal, possibly identifying periodic digits. For 'decimal, if it's not possible to render the number as a decimal exactly, a fraction representation might be generated. For 'mixed an improper fraction representation might be generated if a mixed representation could not be read back.

If *fraction-view* is #f, this option comes from some unspecified user preference.

```
(horizontal markup ...) → markup?
  markup : markup?
```

This procedure arranges the *markup* arguments horizontally.

```
(vertical markup ...) → markup?
  markup : markup?
```

This procedure arranges the *markup* arguments vertically.

```
(transform-markup mappers markup) → markup?
  mappers : (listof pair?)
  markup : markup?
```

This is the same as transform-markup.

```
(markup-transform-image-data transform-image-data
                             markup)              → markup?
  transform-image-data : (any/c . -> . any/c)
  markup : markup?
```

This walks over a markup tree, leaving everything unchanged except image-markup values. For those, it applies *transform-image-data* to its datafield, replacing it by the return value.

3 Rendering Markup to Text

```
(require simple-tree-text-markup/text)
      package: simple-tree-text-markup-lib
```

This module renders markup to text by printing to a port.

```
(display-markup markup [output-port]) → any
  markup : markup?
  output-port : output-port? = (current-output-port)
```

Renders a textual version of *markup* to *output-port*. It uses Unicode lines and corners to display framed markup.

```
(number-markup->string number
                       [#:exact-prefix exact-prefix
                        #:inexact-prefix inexact-prefix
                        #:fraction-view fraction-view])
 → string?
  number : number?
  exact-prefix : (or/c 'always 'never 'when-necessary) = 'never
  inexact-prefix : (or/c 'always 'never 'when-necessary)
                 = 'never
  fraction-view : (or/c #f 'mixed 'improper 'decimal) = #f
```

This is a convenience function that generates a textual number representation according to the specification of number-markup.

4 Generating Markup From a Port

```
(require simple-tree-text-markup/port)
      package: simple-tree-text-markup-lib
```

This modules define procedures for creating output ports whose output is captured as a markup object.

```
(make-markup-output-port special->markup)
 → output-port? (-> markup?)
  special->markup : (any/c . -> . markup?)
```

This procedure returns an output port and a thunk.

The thunk will return whatever has been output to the port as a markup object.

The port also supports `write-special`: Any object output through it will be converted into markup by the *special->markup* procedure.

```
(make-markup-output-port/unsafe special->markup)
 → output-port? (-> markup?)
  special->markup : (any/c . -> . markup?)
```

Thread-unsafe version of `make-markup-output-port`.

```
srclocs-special<%> : interface?
```

This interface is for implementation by objects written via `write-special` to a port created by the procedures above: It marks objects (typically snips) that represent a sequence of source locations, for which the markup output should render a link.

Note that, in order to make use of this, you will need to call `make-markup-output-port` with a `special->markup` argument that looks for specials implementing this interface and converts them to markup appropriately.

```
(send a-srclocs-special get-srclocs)
 → (or/c #f (listof srcloc?))
```

> Returns the source locations represented by the special object, most relevant first in the list.