# TeXmacs Scheme developer guide

# Table of contents

# Chapter 1

# Overview of the Scheme extension language

One major characteristic of TeXmacs is the possibility to extend the editor using the Guile-Scheme *extension language*. Such extensions can be simple, like a personal boot file containing frequently used keyboard shortcuts, or more complex, like a plug-in with special editing routines for a particular type of documents. The Scheme language can also be used interactively from within the editor or invoked by special markup like "actions".

In this chapter, we give an overview of why and how to use Scheme from within TeXmacs. The first sections provide sufficient information for someone who wants to program some basic customization of the keyboard and menus. The latter sections give an introduction to the general architecture of the Scheme API and some important features and particularities of way Scheme is used within TeXmacs. The reading of the overview is highly recommended to anyone who wants to make non-trivial use of Scheme inside TeXmacs.

More complete documentation about the Scheme modules provided by TeXmacs is available from the Help→Scheme extensions menu. We also recommend the following on-line manuals about Scheme and its Guile implementation:

- The Scheme programming language.
- Guile reference manual.

For further information about Scheme, we refer to `http://www.schemers.org`. As a general rule, we also encourage users to take a look at the TeXmacs source code for concrete examples on how to use Scheme from within TeXmacs.

## 1.1. Why TeXmacs uses Scheme as its extension language

At a first glance, the choice of Scheme as an extension language for TeXmacs may seem a bit strange for people who are accustomed to more conventional programming languages, such as C++, Java or Python. In particular, its heavy use of parenthesis frightens more than one person.

Our choice of Scheme has been motivated by the fact that the language is highly flexible in several ways:

1. It is easy to mix programs and data in a common framework.
2. It is easy to customize the language itself, by adding new programming constructs.
3. It is easy to write programs on a very abstract level.

The first two features are very particular important for extension languages. Indeed, one major use of extension languages is to store data for the application (like keyboard shortcuts and menus) in an intelligent way. Furthermore, the application usually provides some very typical features, which may need to be reflected at the level of the extension language.

For the first two features, the simplicity of the parenthesized notation used by SCHEME is also an advantage. Indeed, consider the following fragment of the definition of the File menu:

```
(menu-bind file-menu
  ("New" (new-buffer))
  ("Load" (choose-file load-buffer "Load file" ""))
  ("Save" (save-buffer))
  ...)
```

The entries of the menu (the data) and the corresponding actions (the programs) are very readable using the bracket notation. Similarly, when defining a new language primitive, the systematic use of the bracket notation relieves the user from the burden of making the corresponding changes in the parser.

## 1.2. When and how to use Scheme

You may invoke SCHEME programs from TEXMACS in different ways, depending on whether you want to customize some aspects of TEXMACS, to extend the editor with new functionality, to make your markup more dynamic, and so on. In this section, we list the major ways to invoke SCHEME routines.

**User provided initialization files.**

In order to customize the basic aspects of TEXMACS, you may provide one or both of the initialization files

```
~/.TeXmacs/progs/my-init-texmacs.scm
~/.TeXmacs/progs/my-init-buffer.scm
```

The file `my-init-texmacs.scm` is loaded when booting TEXMACS and `my-init-buffer.scm` is booted each time you open a file.

Usually, the file `my-init-texmacs.scm` contains personal keyboard bindings and menus. For instance, when putting the following piece of code in this file, the keyboard shortcuts `⇧TH.` and `⇧PROP.` for starting a new theorem resp. proposition:

```
(kbd-map
  ("D e f ." (make 'definition))
  ("L e m ." (make 'lemma))
  ("P r o p ." (make 'proposition))
  ("T h ." (make 'theorem)))
```

Similarly, the following command extends the standard Insert menu with a special section for the insertion of greetings:

```
(menu-bind insert-menu
  (former)
  ---
  (-> "Opening"
      ("Dear Sir" (insert "Dear Sir,"))
      ("Dear Madam" (insert "Dear Madam,")))
  (-> "Closing"
      ("Yours sincerely" (insert "Yours sincerely,"))
      ("Greetings" (insert "Greetings,"))))
```

The customization of the keyboard and menus is described in more detail in the chapter about the TEXMACS extensions of Scheme. Notice also that, because of the lazy loading mechanism, you can not always assume that the standard key-bindings and menus are loaded before `my-init-texmacs.scm`. This implies that some care is needed in the case of redefinitions.

The file `my-init-buffer.scm` can for instance be used in order to automatically select a certain style when starting a new document:

```
(if (not (buffer-has-name? (current-buffer)))
    (begin
      (init-style "article")
      (buffer-pretend-saved (current-buffer))))
```

Notice that the "no name" check is important: when omitted, the styles of existing documents would also be changed to `article`. The function `buffer-pretend-saved` is used in order to avoid TEXMACS to complain about unsaved documents when leaving TEXMACS without changing the document.

Another typical use of `my-init-buffer.scm` is when you mainly want to use TEXMACS as a front-end to another system. For instance, the following code will force TEXMACS to automatically launch a Maxima session for every newly opened document:

```
(if (not (buffer-has-name? (current-buffer)))
    (make-session "maxima" (url->string (current-buffer))))
```

Using `(url-> tring (current-buffer))` as the second argument of `make-session` ensures that a different session will be opened for every new buffer. If you want all buffers to share a common instance of Maxima, then you should use `"default"` instead, for the second argument.

### User provided plug-ins.

The above technique of Scheme initialization files is sufficient for personal customizations of TEXMACS, but not very convenient if you want to share extensions with other users. A more portable way to extend the editor is therefore to regroup your Scheme programs into a *plug-in*.

The simplest way to write a plug-in *name* with some additional Scheme functionality is to create two directories and a file

`~/.TeXmacs/plugins/`*name*

`~/.TeXmacs/plugins/`*`name`*`/progs`
`~/.TeXmacs/plugins/`*`name`*`/progs/init-`*`name`*`.scm`

Furthermore, the file `init-`*`name`*`.scm` should a piece of configuration code of the form

```
(plugin-configure name
  (:require #t))
```

Any other SCHEME code present in `init-`*`name`*`.scm` will then be executed when the plug-in is booted, that is, shortly after TEXMACS is started up. By using the additional `(:prioritary #t)` option, you may force the plug-in to be loaded earlier during the boot procedure.

Of course, the plug-in mechanism is more interesting when the plug-in contains more than a few customization routines. In general, a plug-in may also contain additional style files or packages, scripts for launching extern binaries, additional icons and internationalization files, and so on. Furthermore, SCHEME extensions are usually regrouped into SCHEME modules in the directory

`~/.TeXmacs/plugins/`*`name`*`/progs`

The initialization file `init-`*`name`*`.scm` should then be kept as short as possible so as to save boot time: it usually only contains lazy declarations which allow TEXMACS to load the appropriate modules only when needed.

For more information about how to write plug-ins, we refer to the corresponding chapter.

### Interactive invocation of SCHEME commands.

In order to rapidly test the effect of SCHEME commands, it is convenient to execute them directly from within the editor. TEXMACS provides two mechanisms for doing this: directly type the command on the footer using the ⌘⇧X shortcut, or start a SCHEME session using Insert→Session→Scheme.

The first mechanism is useful when you do not want to alter the document or when the current cursor position is important for the command you wish to execute. For instance, the command `(inside? 'theorem)` to test whether the cursor is inside a theorem usually makes no sense when you are inside a session.

SCHEME sessions are useful when the results of the SCHEME commands do not fit on the footer, or when you want to keep your session inside a document for later use. Some typical commands you might want to use inside a SCHEME session are as follows (try positioning your cursor inside the session and execute them):

```
scheme] (define (square x) (* x x))
scheme] (square 1111111)
scheme] (kbd-map ("h i ." (insert "Hi there!")))
scheme] ;; try typing ``hi.''
```

### Command-line options for executing SCHEME commands.

TEXMACS also provides several command-line options for the execution of SCHEME commands. This is useful when you want to use TEXMACS as a batch processor. The SCHEME-related options are the following:

`-x` *`cmd`*.
Executes the scheme command *`cmd`* when booting has completed. For instance,

```
texmacs -x "(display "Hi there\n")"
```

causes TeXmacs to print "Hi there!" when starting up. Notice that the `-x` option may be used several times.

`-q`.
This option causes TeXmacs to quit. It is usually used after a `-x` option. For instance,

```
texmacs text.tm -x "(print)" -q
```

will cause TeXmacs to load the file `text.tm`, to print it, and quit.

`-c` *in* *out*.
This options may be used to convert the input file *in* into the output file *out*. The suffixes of *in* and *out* determine their file formats.

**Invoking Scheme scrips from TeXmacs markup.**

TeXmacs provides two major tags for invoking Scheme scripts from within the markup:

⟨action|*text*|*script*⟩.
This tag works like a hyperlink with body *text*, but such that the Scheme command *script* is invoked when clicking on the *text*. For instance, when clicking here, you will launch an `xterm`.

⟨extern|*fun*|*arg-1*|...|*arg-n*⟩.
This tag is used in order to implement macros whose body is written in Scheme rather than the TeXmacs macro language. The first argument *fun* is a scheme function with *n* arguments. During the typesetting phase, TeXmacs passes the arguments *arg-1* until *arg-n* to *fun*, and the result will be typeset. For instance, the code

```
⟨extern|(lambda (x) `(concat "Hallo " ,x))|Piet⟩
```

produces the output "Hallo Piet". Notice that the argument "Piet" remains editable.

It should be noticed that the direct invocation of Scheme scripts from within documents carries as risk: an evil person might send you a document with a script which attempts to erase your hard disk (for instance). For this reason, TeXmacs implements a way to test whether scripts can be considered secure or not. For instance, when clicking here (so as to launch an `xterm`), the editor will prompt you by default in order to confirm whether you wish to execute this script. The desired level of security can be specified in Edit→Preferences→Security. When writing your own Scheme extensions to TeXmacs, it is also possible to define routines as being secure.

## 1.3. General architecture of the Scheme API

When programming Scheme extensions of TeXmacs, it may be useful to be conscious of the internal architecture of the Scheme modules inside TeXmacs (see figure 1.1).

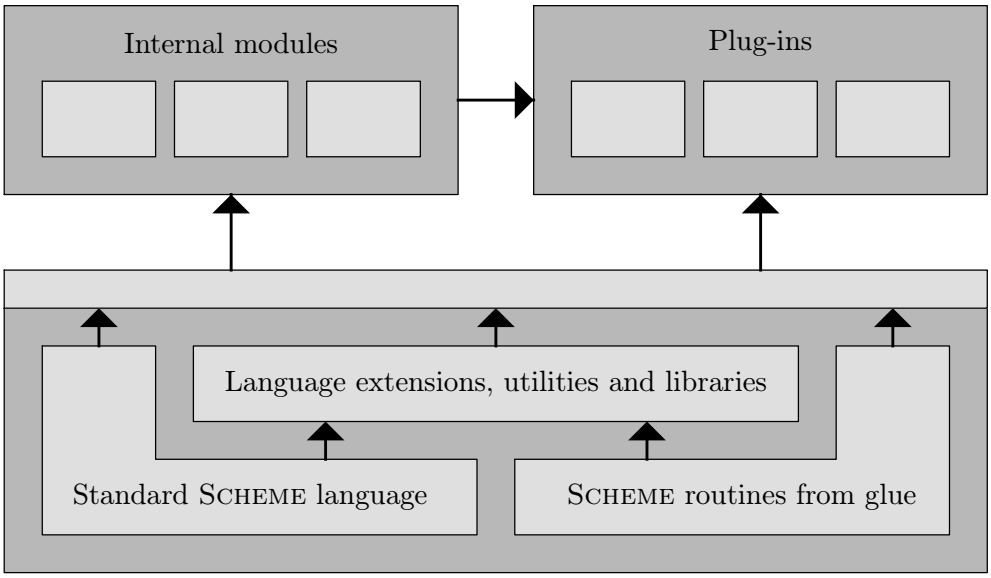



**Figure 1.1.** Schematic organization of the SCHEME API.

### Built-in SCHEME commands.

On the very basic level, one has the standard SCHEME language, with some enhancements by the GUILE implementation (these extensions are used as least as possible, for future portability). The standard SCHEME language is enriched by some routines implemented in the C++ part of TEXMACS and exported to SCHEME via the glue. If you unpacked the source code of TEXMACS in *source-dir*, then you can find a full list of the routines exported by the glue in the files

```
source-dir/src/Guile/Glue/build-glue-base.scm
source-dir/src/Guile/Glue/build-glue-editor.scm
source-dir/src/Guile/Glue/build-glue-server.scm
```

### Extensions to SCHEME and further utilities.

Above the standard SCHEME language and the extra routines from the glue, TEXMACS comes with a second level of language extensions, utilities and libraries. The corresponding SCHEME files can be found in the directories

```
$TEXMACS_PATH/progs/kernel
$TEXMACS_PATH/progs/utils
```

Roughly speaking, the functionality provided by this second level is the following:

- A certain number of frequently used abbreviations, like `==` for `equal?`.
- General language extensions for contextual overloading, logical programming, etc.
- TEXMACS-specific language extensions for the definition of menus, keyboard shortcuts, etc.

- Additional routines for TEXMACS content manipulation and pattern matching.
- Further utilities and libraries for common types like strings and lists.

Whereas the modules in `$TEXMACS_PATH/progs/kernel` are automatically loaded, all modules in `$TEXMACS_PATH/progs/utils` have to be explicitly included.

**Internal modules and plug-ins.**

The remaining SCHEME extensions of TEXMACS are regrouped into *internal modules* which usually correspond to a particular type of content. For instance, the directories

```
$TEXMACS_PATH/progs/source
$TEXMACS_PATH/progs/math
$TEXMACS_PATH/progs/table
```

respectively contain routines for editing source code, mathematics and tables. Exceptions are the internal modules `content` and `fonts`, which rather correspond to a particular type of functionality. Each internal module corresponds to a group of files, each of which corresponds to an individual *TEXMACS module*. The internal modules are designed to be as independent as possible.

From the SCHEME point of view, the structure of a plug-in is very similar to that of an internal module. Each plug-in defines a collection of SCHEME programs in its `progs` subdirectory. Although distinct plug-ins may in principle depend on each other, they are usually designed in a way which makes them as independent as possible.

## 1.4. The module system and lazy definitions

As explained above, each SCHEME file inside TEXMACS or one of its plug-ins corresponds to a *TEXMACS module*. The individual TEXMACS modules are usually grouped together into an internal or external module, which corresponds to a directory on your hard disk.

Any TEXMACS module should start with an instruction of the form

```
(texmacs-module name
  (:use submodule-1 ... submodule-n))
```

The *name* of the module is a list which corresponds to the location of the corresponding file. More precisely, TEXMACS searches for its modules in the path `$GUILE_LOAD_PATH`, which defaults to the standard GUILE load path, combined with `$TEXMACS_PATH/progs` and all `progs` subdirectories in the plug-ins. For instance, the module `(math math-edit)` corresponds to the file

```
$TEXMACS_PATH/progs/edit/math-edit.scm
```

The user should explicitly specify all submodules on which the module depends, except those modules which are loaded by default, i.e. all language extensions and utilities in the directories

```
$TEXMACS_PATH/progs/kernel
$TEXMACS_PATH/progs/utils/library
```

All symbols which are defined inside the module using `define` or `define-macro` are only visible within the module itself. In order to make the symbol publicly visible you should use `tm-define` or `tm-define-macro`. Currently, because of implementation details for the contextual overloading system, as soon as a symbol is declared to be public, it becomes visible inside all other modules. However, you should not rely on this: in the future, explicit importation with `:use` might become necessary.

Because the number of TEXMACS modules and plug-ins keeps on growing, it is inefficient to load all modules when booting. Instead, initialization files are assumed to declare the provided functionality in a *lazy* way: whenever the functionality is explicitly needed, TEXMACS is triggered to load the corresponding modules (if this was not already done). In addition, TEXMACS may load some of these modules during spare time, when the computer is waiting for user input. Indeed, this helps increasing the reactivity of TEXMACS at the first use of the functionality.

For instance, assume that you defined a large new editing function `foo-action` inside the module `(foo-edit)`. Then your initialization file `init-foo.scm` would typically contain a line

```
(lazy-define (foo-edit) foo-action)
```

Similarly, lazy keyboard shortcuts and menus for `foo` might be defined using

```
(lazy-keyboard (foo-kbd) in-foo-mode?)
(lazy-menu (foo-menu) foo-menu)
```

For more concrete examples, we recommend the user to take a look at the standard initialization file `init-texmacs.scm`.

On the negative side, the mechanism for lazy loading has the important consequence that you can no longer make assumptions on when a particular module is loaded. For instance, when you attempt to redefine a keyboard shortcut in your personal initialization file, it may happen that the standard definition is loaded after your "redefinition". In that case, your redefinition remains without consequence.

For this reason, TEXMACS also provides the instruction `import-from` to force a particular module to be loaded. Similarly, the commands `lazy-keyboard-force`, `lazy-plugin-force`, etc. may be used to force all lazy keyboard definitions resp. plug-ins to be loaded. In other words, the use of laziness forces to make implicit dependencies between modules more explicit.

In the case when you want to redefine keyboard shortcuts, the contextual overloading system gives you an even more fine-grained control. For instance, assume that the keyboard shortcut `XXX` has been defined twice, both in general and in math mode. After calling `lazy-keyboard-force` and overriding the general definition of the shortcut, the special definition will still take precedence in math mode. Alternatively, you may redefine the keyboard shortcut using

```
(kbd-map
  (:mode prevail?)
  ("x x x" action))
```

This redefinition will prevail both in general and in math mode.

## 1.5. Contextual overloading

For large software projects, it is important that different modules can be developed as independently as possible one from each other. Furthermore, fundamental modules often implement default behaviour which is to be overwritten in a more specialized module. In order to facilitate these two requirements, TeXmacs implements a system of *contextual overloading*.

In order to get the main idea behind this system, consider the implementation of a given functionality, like hitting the return key. Depending on the context, different actions have to be undertaken: by default, we start a new paragraph; inside a table, we start a new row; etc. A naive implementation would check all possible cases in a routine `kbd-enter` and call the corresponding routine. However, this makes it impossible to add a new case in a new module without modifying the module which defines `kbd-enter`. By contrast, the system of contextual overloading allows the user to *conditionally* redefine the routine `kbd-enter` several times in distinct modules.

For instance, assume that we want to define a function `hello` which inserts "Hello" by default, but "hello()" in mode math, while positioning the cursor between the brackets. Using contextual overloading, this may be done as follows:

```
(tm-define (hello) (insert "Hello"))
(tm-define (hello) (:require (in-math?)) (insert-go-to "hello()" '(6)))
```

The order in which routines are overloaded is important. TeXmacs first tries the latest (re)definition. If this definition does not satisfy the requirements (`(in-math?)`, in our case), then it tries the before last (re)definition, and so on until an implementation is found which matches the requirements. For example, if we invert the two declarations in the above example, then the general unconditional definition of `hello` will always prevail. If the two declarations are made inside different modules, then it is up to the user to ensure that the modules are loaded in an appropriate order.

Inside a redefinition, it is also possible to access the former definition using the keyword `former`. In particular, the code

```
(tm-define (hello)
  (if (in-math?) (insert-go-to "hello()" '(6)) (former)))
```

is equivalent to the second declaration in our example.

Contextual overloading generalizes more classical overloading on the types of the arguments, such as C++ style polymorphism. Although one may overload on the types of the arguments, it is also possible to impose more general conditions on the arguments. For instance, one may sometimes wish to write the following kind of code:

```
(tm-define (my-replace what by)
  default-implementation)

(tm-define (my-replace what by)
  (:require (== what by))
  (noop))
```

Besides `tm-define`, several other added language primitives support the contextual overloading mechanism. For instance, `kbd-map` and `menu-bind` support overloading on mode. The `tm-define-macro` and `tm-property` primitives are analogous to `tm-define`.

## 1.6. META INFORMATION AND LOGICAL PROGRAMMING

Small software projects usually consist of a collection of routines and data. In a large software project, where a typical contributor has no complete overview of the program, it is a good practice to associate additional *meta-information* to the individual routines and data. This meta-information typically serves documentation purposes, but becomes even more interesting if it can be used in an automated fashion to implement more general additional functionality.

The `tm-define` macro supports several options for associating meta-information to SCHEME functions and symbols. For instance, the `:synopsis`, `:argument` and `:returns` options allow you to associate short documentation strings to the function, its arguments and its return value:

```
(tm-define (square x)
  (:synopsis "Compute the square of @x")
  (:argument x "A number")
  (:returns "The square of @x")
  (* x x))
```

This information is exploited by TEXMACS in several ways. For instance, the synopsis of the function can be retrieved by executing `(help square)`. More interestingly, assuming that we defined `square` as above, typing ⌘X followed by `square` and ↵ allows you to execute `square` in an interactive way: you will be prompted for "A number" on the footer. Moreover, after typing ⌘X, you will be able to use "tab-completion" in order to enter `square`: typing `SQU`⇥ will usually complete into `square`.

In a similar vein, the `:interactive` and `:check-mark` options allow you to specify that a given routine requires interactive user input or when it should give rise to a check-mark when used in a menu. For instance, the statement

```
(tm-property (choose-file fun text type)
  (:interactive #t))
```

in the source code of TEXMACS states that `choose-file` is an interactive command. As a consequence, the File→Load entry, which is defined by

```
("Load" (choose-file load-buffer "Load file" ""))
```

will be followed by dots ... in the File menu. The interesting point here is that, although the command `choose-file` may be reused several times in different menu entries, we only have to specify once that it is an interactive command. Similarly, consider the definition

```
(tm-define (toggle-session-math-input)
  (:check-mark "v" session-math-input?)
  (session-use-math-input (not (session-math-input?))))
```

Given a menu item with (toggle-session-math-input) as its associated action, this definition specifies in particular that a check-mark should be displayed before the menu item whenever the session-math-input? predicate holds.

Another frequently used option is :secure, which specifies that a given routine can be used inside TEXMACS documents, in particular inside extern and action macros. For instance, the default implementation of the fold tag allows the user to click on the "○" before the folded text so as to unfold the tag. When doing this, the scheme script mouse-unfold is launched. However, for this to work, the mouse-unfold function needs to be secure:

```
(tm-define mouse-unfold
  (:secure #t)
  (with-action t
    (tree-go-to t :start)
    (fold)))
```

You can read more about the tags which depend on SCHEME scripts in "Invoking SCHEME scripts from TEXMACS markup".

In the future, the number of options for entering meta-information is likely to increase. TEXMACS also supports an additional mechanism for the automatic deduction of new meta-properties from existing meta-properties. This mechanism is based on a less general, but more efficient form of *logical programming*. However, since it is not fully stable yet, it will be documented only later.

## 1.7. THE TEXMACS CONTENT MODEL

All TEXMACS documents or document fragments can be thought of as *trees*, as explained in more detail in the chapter about the TEXMACS document format. Inside SCHEME programs, there are two main ways to represent such trees, depending on whether one manipulates active or passive documents:

**Passive documents and SCHEME trees.**

Passive documents, like those which are processed by a conversion tool, are usually represented by *scheme trees*. For instance, the fraction

$$\frac{a^2}{b+c}$$

is typically represented by

```
(frac (concat "a" (rsup "2")) "b+c")
```

This representation is convenient in the sense that they can be manipulated directly using standard SCHEME routines on lists.

**Active documents and C++ trees.**

Active documents, like ones which are visible in one of the editors windows, are rather represented using the internal C++ type `tree`, which has been exported to SCHEME via the glue. When a tree is part of a real document inside the editor, the tree is aware about its position inside the document. Using routines from the tree API, you may then make changes in the document simply by assigning new values to the tree.

For instance, consider the following experiment: open two windows and start a SCHEME session in each window. In the second window, enter the lines

```
scheme] (use-modules (utils library tree))
scheme] (define t (buffer-tree))
```

In the first window, you may now modify the document in the second window using commands like

```
scheme] (tree-set! t (tree 'document (string->tree "First line.")
                                     (string->tree "Second line.")))
scheme] (tree-set t 1 (string->tree "New second line."))
scheme] (tree-set t 0 (tree 'strong (tree-ref t 0)))
```

**A common framework.**

From the last three lines in above experiment, it becomes apparent that it is quite cumbersome to manipulate trees using the standard tree constructors. For this reason, TEXMACS provides a hybrid type `content` for manipulating scheme trees and C++ trees in a common framework. For instance, the last three lines in the above experiment may be replaced by

```
scheme] (tree-set! t '(document "First line." "Second line."))
scheme] (tree-set t 1 "New second line.")
scheme] (tree-set t 0 `(strong ,(tree-ref t 0)))
```

More precisely, a scheme expression of the type `content` is either a string, a tree or a list whose first element is a symbol and whose remaining elements are other expressions of type `content`. TEXMACS provides several routines (usually prefixed by `tm-`) for basic operations on content, like `tm-car`, `tm-arity`, `tm->list`, `tm-equal?`, etc. Most higher level routines are built on top of these routines, so as to accept arguments of type `content` whenever appropriate.

**Persistent positions inside trees.**

Besides the fact that trees remember their *positions* inside the global edit tree, it is also possible to create cursor positions inside the global edit tree, which are naturally updated when modifications take place. This technique is useful when you want to write an editing routine which does not act locally at the cursor position. For instance, the following routine can be used to insert content at the start of the current buffer in a reliable way:

```
(define (insert-at-buffer-start t)
  (with-cursor (path-start (root-tree) (buffer-path))
    (insert t)))
```

The `with-cursor` macro temporarily changes the cursor position, while storing the old cursor position in such a way that it will be updated during changes of the document. The user may also use the more explicit routines `position-new`, `position-delete`, `position-set` and `position-get` to manage persistent positions.

## 1.8. STANDARD UTILITIES

Besides the basic concepts from the previous sections, which underly the scheme API for TEXMACS, the SCHEME kernel implements several other utilities and language extensions. In this section, we will briefly sketch some of them on hand of examples. Further details can be found in the chapter about TEXMACS extensions to SCHEME and utilities.

### Regular expressions.

TeXmacs implements the routines `match?` and `select` for matching regular expressions and selecting subexpressions along a "path". These routines both work for the `content` type. For instance, in order to search all expressions of the form

$$\frac{a}{1+\sqrt{b}}$$

in the current buffer, where $a$ and $b$ are general expressions, one may use the following Scheme command:

```
scheme] (select (buffer-tree) '(:* (:match (frac :%1 (concat "1+" (sqrt
        :%1))))))
```

### Dialogues.

TeXmacs supports several commands for interactive dialogues with the user. For instance, when executing the following scheme command, you will be prompted for two numbers, whose product will be displayed in the footer:

```
Scheme] (user-ask "First number:"
          (lambda (a)
            (user-ask "Second number:"
              (lambda (b)
                (set-message (number->string (* (string->number a)
                                                (string->number b)))
                             "product")))))
Scheme]
```

### User preferences.

When writing a plug-in, you may wish to define some new user preferences. This can be done using the `define-preferences` command, which adds a list of user preferences, together with their default values and a call-back routine. The call-back routine is called whenever you change the corresponding preference. For instance:

```
(define-preferences
  ("Gnu's hair color" "brown" notify-gnu-hair-change)
  ("Snail's cruising speed" "1mm/sec" notify-Achilles))
```

Preferences can be set, reset and read using `set-preference`, `reset-preference` and `get-preference`.

### New data formats and converters.

New data formats and converters can be declared using the `define-format` and `converter` instructions. When a format can be converted from or into TeXmacs, then it will automatically appear into the File→Export and File→Import menus. Similarly, when a format can be converted to Postscript, then it also becomes a valid format for images. TeXmacs also attempts to combine explicitly declared converters into new ones.

Typically, the declaration of a new format and a converter would look like:

```
(define-format blablah
  (:name "Blablah")
  (:suffix "bla"))

(converter blablah-file latex-file
  (:require (url-exists-in-path? "bla2tex"))
  (:shell "bla2tex" from ">" to))
```

# Chapter 2

# TeXmacs extensions to Scheme and utilities

## 2.1. TeXmacs abbreviations

## 2.2. Matching regular expressions

Regular expressions naturally generalize from strings to trees and allow to test whether a given tree matches a given pattern. TeXmacs implements the primitives `match?` and `match` for this purpose, which also provide support for wildcards, user-defined grammars and more.

`(match? expr pattern)` (check whether a scheme expression satisfies a pattern)

This function determines whether a scheme expression `expr` satisfies a given `pattern`. It will be detailed below how to form valid patterns. The pattern may contain named wildcards, in case of success, we return a list with matches for these wildcards. In case of failure, we return `#f`. The expression `expr` may contain trees, in which case we understand that such tree subexpressions should match their scheme counterparts. For instance, `(match? (tree "x") "x")` will return `(())`, whereas `(match? (tree "x") "y")` returns `#f`.

`(match l pattern bindings)` (solutions to a given pattern under bindings)

Given a list `l` of scheme expressions, a `pattern` with free variables and an association list of `bindings`, this routine determines all substitutions of free variables by values (extending the given `bindings`), for which `l` matches the `pattern`.

`(define-regexp-grammar rules*)` (user defined matching grammars)

Given a list of rules of the form `(:var pattern-1 ... pattern-n)`, this instruction defines a new terminal symbol `:var` for each such rule, which matches the disjunction of the patterns `pattern-1` until `pattern-n`. This terminal symbol can then be used as an abbreviation in matching patterns. Grammar rules may be interdependent. See example below.

Valid patterns are formed in the following ways:

`leaf` (symbols, strings, etc.)

A `leaf` is only matched against itself.

`(pattern-1 ... pattern-n)` (lists)

In the case when lists `l-1` until `l-n` match `pattern-1` until `pattern-n`, their concatenation matches the pattern `(pattern-1 ... pattern-n)`.

`:%1`, `:%2`, `:%3` ..., `:*` (wildcards)

The wildcard `:%n`, where `n` is a number matches any list of length `n`. The wildcard `:*` matches any list, including the empty list.

`'var` (variables)

This pattern attempts to bind the variable `var` against the expression. If `var` is used only once, then it essentially behaves as a wildcard. More generally, it can be used to form patterns with identical subexpressions. For instance, the pattern `(frac 'x 'x)` will match all fractions $\frac{x}{x}$.

`:var` (user-provided grammar rules)

In the case when `:var` is a user-provided terminal symbol (see `define-regexp-grammar` above), this pattern matches the corresponding grammar.

`:pred?` (arbitrary SCHEME predicates)

Given a SCHEME predicate `pred?`, such as `string?`, this pattern matches any scheme expression which satisfies the predicate.

```
(:not pattern)
(:or pattern-1 ... pattern-n)
(:and pattern-1 ... pattern-n)
```
(logical operations)

Negation, disjunction and conjunction of patterns.

`(:repeat pattern)` (repetition)

Given lists `l-1` until `l-n` which match `pattern`, their concatenation matches the repetition `(:repeat pattern)`. In particular, the empty list is matched.

`(:group pattern-1 ... pattern-n)` (grouping)

Groups a concatenation of patterns into a new list patterns. For instance, all lists of the form `(a b a b ... a b)` are matched by `(:repeat (:group a b))`, whereas `(:repeat (a b))` rather matches all lists of the form `((a b) (a b) ... (a b))`.

`(:quote expr)` (quotation)

Only matches a given expression `expr`.

**Example 2.1.** The tree

```
(define t '(foo (bar "x") (bar "y") (option "z")))
```

matches the pattern `(foo (:repeat (bar :%1)) :*)`, but not `(foo (:repeat (bar 'x)) :*)`. The call `(match t '(foo 'x 'y :*))` will return `(((x . (bar "x")) (y . (bar "y"))))`. Notice that `(x . (bar "x"))` will be displayed as `(x bar "x")`:

```
Scheme] (define t '(foo (bar "x") (bar "y") (option "z")))
Scheme] (match? t '(foo 'x 'y :*))
   (((y bar "y") (x bar "x")))
```

**Example 2.2.** Consider the grammar

```
(define-regexp-grammar
  (:a a b c)
  (:b (:repeat :a)))
```

Then the list `(a b x y c a a)` matches the pattern `(:b :%2 :b)`.

## 2.3. Selection of subexpressions

Besides pattern matching on trees, TeXmacs provides the routine `select` for pattern matching along paths. Given a tree, this mechanism typically allows the user to select all subtrees which are reached following a path which meets specific criteria. For instance, one might to select the second child of the last child or all square roots inside numerators of fractions. The syntax of the selection patterns is also used for high level tree accessors.

`(select expr pattern)` (select subexpressions following a pattern)

Select all subtrees inside a hybrid tree `expr` according to a specific path `pattern`.

Patterns are lists of atomic patterns of one of the following forms:

`0`, `1`, `2`, ... (select a specific child)

Given an integer `n`, select the `n`-th child of the input tree. For instance, `(select '(frac "1" "2") '(0))` returns `("1")`.

`:first`, `:last` (select first or last child)

Select first or last child of the input tree.

`(:range start end)` (select children in a range)

Select all children in a specified range.

`label` (select children with a given label)

Select all compound subtrees with the specified `label`. Example:

```
Scheme] (select '(document (strong "x") (math "a+b") (strong "y"))
        '(strong))
   ((strong "x") (strong "y"))
```

`:%1`, `:%2`, `:%3`, ... (select descendants of a given generation)

The pattern `:%n`, where `n` is a number, selects all descendants of the `n`-th generation. Example:

```
Scheme] (select '(foo (bar "x" "y") (slash (dot))) '(:%2))
   ("x" "y" (dot))
```

`:*` (select all descendants)

This pattern selects all descendants of the tree. For instance, `(select t '(:* frac 0 :* sqrt))` selects all square roots inside numerators of fractions inside `t`.

`(:match pattern)` (matching)

This pattern matches the input tree if and only the input tree matches the specified `pattern` according to `match?`. Example:

```
Scheme] (select '(foo "x" (bar)) '(:%1 (:match :string?)))
   ("x")
```

Example with creation of a custom predicate:

```
Scheme] (select '(foo "x" (bar)) '(:* (:match :tree-atomic?)))
   ()
Scheme]
```

List of useful predicates:

`(:or pattern-1 ... pattern-n)`
`(:and pattern-1 ... pattern-n)` (boolean expressions)

These rules allow for the selection of all subtrees which satisfy one among or all patterns `pattern-1` until `pattern-n`.

In the case when the input tree is active, the function `select` supports some additional patterns which allow the user to navigate inside the tree.

`:up` (parent)

This pattern selects the parent of the input tree, if it exists.

`:down` (child containing the cursor)

If the cursor is inside some child of the input tree, then this pattern will select this child.

`:next` (next child)

If the input tree is the $i$-th child of its parent, then this pattern will select the $(i+1)$-th child.

`:previous` (previous child)

If the input tree is the $i$-th child of its parent, then this pattern will select the $(i-1)$-th child.

## 2.4. Logical programming extensions

## 2.5. Function definition and contextual overloading

Conventional programming languages often provide some means to overload certain functions depending on the types of the arguments. TEXMACS provides additional context-based overloading mechanisms, which require the use of the `tm-define` construct for function definitions (and `tm-define-macro` for macro definitions). Definition with `tm-define` also allows the specification of properties of the function/macro: arguments, synopsis, etc.

Furthermore, one may use `tm-property` for associating additional properties, such as interactivity or default values for the arguments, of a function *which is already defined*, specifically functions exported from C++ code through the glue.

`(tm-define head options* body*)` (TEXMACS function definition)

`(tm-define-macro head options* body*)` (TEXMACS macro definition)

TEXMACS function and macro declarations are similar to usual declarations based on `define` and `define-macro`, except for the additional list of `options` and the fact that all functions and macros defined using `tm-define` and `tm-define-macro` are public. Each option is of the form `(:kind arguments*)` and the `body` starts at the first element of the list following `head` which is not of this form. Available options are `:type`, `:synopsis`, `:returns`, `:note`, `:argument`, `:default`, `:proposals`, `:secure`, `:check-mark`, `:interactive` and `:balloon`.

`(tm-property head options*)` (TEXMACS properties definition)

`tm-property` allows the declaration of TEXMACS properties for functions which have already been defined, specifically for functions exported through the glue. Available options are `:type`, `:synopsis`, `:returns`, `:note`, `:argument`, `:default`, `:proposals`, `:secure`, `:check-mark`, `:interactive` and `:balloon`.

### Contextual overloading.

We will first describe the most important `:require` option for contextual overloading, which was already discussed before.

`(:require cond)` (argument based overloading)

This option specifies that one necessary condition for the declaration to be valid is that the condition `cond` is met. This condition may involve the arguments of the function.

As an example, let us consider the following definitions:

```
(tm-define (special t)
  (and-with p (tree-outer t)
    (special p)))

(tm-define (special)
  (:require (tree-is? t 'frac))
  (tree-set! t `(frac ,(tree-ref t 1) ,(tree-ref t 0))))

(tm-define (special)
  (:require (tree-is? t 'rsub))
  (tree-set! t `(rsup ,(tree-ref t 0))))
```

The default implementation of `special` is to apply `special` to the parent `p` of `t` as long as `t` is not the entire document itself. The two overloaded cases apply when `t` is either a fraction or a right subscript.

Assuming that your cursor is inside a fraction inside a subscript, calling `special` will swap the numerator and the denominator. On the other hand, if your cursor is inside a subscript inside a fraction, then calling `special` will change the subscript into a superscript.

When the conditions of several (re)declarations are met, then the last redeclaration will be used. Inside a redeclaration, one may also use the `former` keyword in order to explicitly access the former value of the redefined symbol.

`(:mode mode)` (mode-based overloading)

This option is equivalent to `(:require (mode))` and specifies that the definition is only valid when we are in a given `mode`. New modes are defined using `texmacs-modes` and modes can inherit from other modes.

`(texmacs-modes . modedefs)` (define new texmacs modes)

Use this macro to define new modes that you can use for contextual overloading, for instance in `kbd-map`. Modes may be made dependent on other modes. This macro takes a variable number of definitions as arguments, each of the form `(mode-name conditions . dependencies)`. End your `mode-name` and any dependencies with one `%`, like this:

```
(texmacs-modes
  (in-verbatim% (inside? 'verbatim) in-text%)
  (in-tt% (inside? 'tt)))
```

When creating new modes remember to place first the faster checks (against booleans, etc.) for speed.

**Other options for function and macro declarations.**

Besides the contextual overloading options, the `tm-define` and `tm-define-macro` primitives admit several other options for attaching additional information to the function or macro. We will now describe these options and explain how the additional information attached to functions can be exploited.

**Warning 2.3.** A current limitation of the implementation is that functions overloaded using `:require` and `:mode` cannot have different options. This means in particular that you cannot specify different values for `:synopsis` depending on the context.

`(:synopsis short-help)` (short description)

This option gives a short description of the function or macro, in the form of a string `short-help`. As a convention, SCHEME expressions may be encoded inside this string by using the `@`-prefix. For instance:

```
(tm-define (list-square l)
  (:synopsis "Appends the list @l to itself")
  (append l l))
```

The synopsis of a function is used for instance in order to provide a short help string for the function. In the future, we might also use it for help balloons describing menu items.

`(:argument var description)`
`(:argument var type description)` (argument description)

This option gives a short `description` of one of the arguments `var` to the function or macro. Such a description is used for instance for the prompts, when calling the function interactively. For these uses, the second format allows for the specification of a `type` which changes how the widgets/prompts work. Some allowed values are `"string"`, the default, and `"file"` and `"directory"`. If any of the last two is specified, tab completion in the interactive prompt will traverse the file system.

`(:returns description)` (return value description)

This option gives a short `description` of the return value of the function or macro.

`(:type (-> from to))` (type conversion description)

This option specifies that a function or macro performs a conversion from the data type `from` to the data type `to`.

## 2.6. Interactive dialogues

## 2.7. User preferences

Preferences are used to store any information you need to keep across different runs of TeXmacs, like window position and size, active menu bars, etc. Internally they are stored in the users home directory as a Scheme list of items like `("name" value)` which therefore has in principle no structure. However, a good practice to avoid conflicts is to prefix your options by the name of the plugin or module you are creating, like in `"gui:help-window-position"`.

The first step in defining a new preference is adding it with `define-preferences` and assigning a call-back function to handle changes in the preference. This is important for instance in menus, where a click on an item simply sets some preference to some value and it's up to the call-back to actually take the necessary actions.

**Warning.** One may not store the boolean values `#t`, `#f` directly into preferences. Instead one should use the strings `"on"` and `"off"`. This is due to the internal storage of default values for preferences using `ahash-table`.

`(define-preferences list)` (define new preferences with defaults and call-backs)

Each element of `list` is of the form `("somename" default-value notify-procedure)` where `notify-procedure` is a procedure taking two arguments like this:

```
(define (notify-procedure property-name value) (do-things))
```

Remember to use the strings `"on"` and `"off"` instead of booleans `#t`, `#f`.

### Example

⇑

```
Scheme] (define (notify-test pref value)
          (display* "Hey! " pref " changed to " value) (newline))
Scheme] (define-preferences ("test:pref" "off" notify-test))
Scheme] (get-preference "test:pref")
   "off"
Scheme] (set-preference "test:pref" "on")
Scheme] (preference-on? "test:pref")
```

```
    #t
Scheme]
```

`(set-preference name value)` (set user preference)

Save preference `name` with value `value`. Then call the call-back associated to this preference, as defined in `define-preferences`.

Remember to use the strings `"on"` and `"off"` instead of booleans `#t`, `#f`.

`(append-preference name value)` (appends a value to the list for a preference)

This convenience function appends `value` to the list of values of preference `name`, or creates a list with one element in case the preference didn't exist. The call-back associated to this preference, as defined in `define-preferences` is called once the modification is done.

`(reset-preference name)` (delete user preference)

Deletes preference `name` from the user preferences.

`(get-preference name)` (get user preference)

Returns the value of preference `name`. If the preference is not defined the string `"default"` is returned.

`(preference-on? name)` (test boolean user preference)

Returns `#t` if the value of preference `name` is `"on"`.

`(toggle-preference name)` (change value of boolean user preference)

Toggles the value of preference `name` between `"on"` and `"off"`.

## 2.8. Adding converters

## 2.9. Keyboard bindings

## 2.10. Defining menus

# Chapter 3

# Programming routines for editing documents

## 3.1. The TEXMACS editing model

Routines for editing documents are usually based on one or several of the following ingredients:

1. Identification of the document fragments which have to be edited.
2. Modification of one or several document fragments.
3. Moving the cursor to a new place.

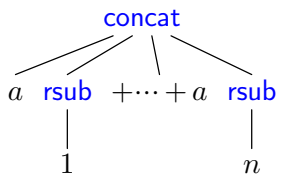
Before going into the precise API which allows you to carry out these tasks, let us first describe the fundamental underlying data types, and go through an example.

**Document fragments.**

All TEXMACS documents or document fragments can be thought of as *trees*, as explained in more detail in the chapter about the TEXMACS document format. For instance, the mathematical formula

$$a_1 + \cdots + a_n \tag{3.1}$$

corresponds to the tree



(3.2)

Trees which are part of a document which is effectively being edited are said to be active, and they are implemented using the Scheme type `tree`.

Besides this representation format, which is preferred when editing document fragments, TEXMACS also allows you to represent passive document fragments by Scheme trees. This alternative representation, which corresponds to the Scheme type `stree`, is more convenient when writing routines for processing documents (such as conversions to another format). Finally, TEXMACS provides a hybrid representation, which corresponds to the Scheme type `content`. The `content` type (corresponding to the prefix `tm-`, for simplicity) is typically used for writing abstract utility routines for trees, which can then be applied indistinctly to objects of type `tree` or `stree`.

One major advantage of active trees (of type `tree`) is that they are aware of their own location in the document. As a consequence, TEXMACS provides editing routines which allow you to modify the document simply by assigning a tree to a different value. For instance, assume that the SCHEME variable `t` contains the subscript 1 in formula (3.1). Then the instruction

```
(tree-set! t "2")
```

will simultaneously change the subscript into a 2 and update the SCHEME variable `t`. Another nicety is that the value of `t` is *persistent* during changes of other parts of the document. For instance, if we change the $a$'s into $b$'s in the formula (3.1), then `t` keeps its value *and* its location. Of course, the location of `t` may be lost when `t` or one of its parents is modified. Nevertheless, the modification routines are designed in such a way that we try hard to remember locations. For instance, if "$a_0+$" is inserted in front of the formula (3.1) using the routine `tree-insert!`, then `t` keeps its value *and* its location, even though one of its ancestors was altered.

Some further precisions and terminology will be useful. First of all, we have seen a distinction between *active* and *passive* trees, according to whether a tree is part of a document or not. Secondly, TEXMACS both supports *native trees* (of type `tree`), which are implemented in C++, and *scheme trees* (of type `stree`), which have a more familiar SCHEME syntax. Finally, *hybrid trees* unify native and scheme trees. Formally speaking, a hybrid tree is either a string, a native tree or a list whose first element is a symbol and whose other elements are again hybrid trees. We notice that active trees are necessarily native, but native trees may both be active or passive. Furthermore, certain descendants of an inactive tree may be active, but we never have the contrary.

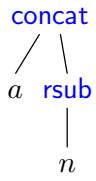
**Positions inside document fragments.**

The main way to address positions inside a tree is via a list of positive integers, called a *path*, and corresponding to the SCHEME type `path`. For instance, assume that `x` corresponds to the expression (3.1). Then the subscript 1 is identified uniquely by the path `(1 0)`. Similarly the cursor position just behind the subscript 1 corresponds to the path `(1 0 1)`. More generally, if `p` is a path to a string leaf, then the path `(rcons p i)` corresponds to the cursor position just behind the `i`-th character in the string (we notice that `rcons` is used to append a new element at the end of a list). If `p` is a path to a non-string subtree, then `(rcons p 0)` and `(rcons p 1)` correspond to the cursor positions before and behind this subtree.

It should be noticed that paths do not necessarily correspond to *valid* subtrees or cursor positions. Clearly, some of the elements in the path may be "out of range". However, certain *a priori* possible cursor positions may correspond to invisible parts of the document (like a cursor position inside a folded argument or an attribute of with). Moreover, two possible cursor positions may actually coincide, like the paths `(0)` and `(0 0)` inside the expression (3.1). In this example, only the second cursor path is valid. Usually, the validity of a cursor path may be quickly detected using DRD (Data Relation Definition) information, which is determined from the style file. In exceptional cases, the validity may only be available after typesetting the document.

It should also be noticed that all active trees are a subtree of the global *TEXMACS edit tree* or *root tree*, which can be retrieved using `(root-tree)`. The routines `tree->path` and `path->tree` can be used in order to get the location of an active tree and the active tree at a given location.

A simple way to address subtrees of a tree in a more persistent way is using object of type `tree`, i.e. by considering the subtrees themselves. The persistent analogue of a cursor path is a *persistent position*, which corresponds to an object of SCHEME type `position`. One particularity of persistent positions is that, even when a tree into which they point is removed, they keep indicating a valid close position in the remaining document. For instance, assume that `pos` stands for the cursor position `(1 0 1)` in the expression (3.1). If we remove $a_1 + \cdots +$, then the tree corresponding to the remaining expression $a_n$ is given by




and the position associated to `pos` becomes `(0 0)`. TEXMACS provides the routines `position-new`, `position-delete`, `position-set` and `position-get` to create, delete, set and get persistent cursor positions.

**Semantic navigation and further utilities.**

Because accessing subtrees using paths may become quite cumbersome, TEXMACS provides some additional functionality to simplify this task. As a general rule, the routines `select` and `match?` may be used to select all subtrees of a given tree which match a certain pattern. For instance, if `x` corresponds to the expression (3.1), then

```
(select x '(rsub :%1))
```

returns a list with the two subscripts 1 and $n$. In fact, `select` may also be used in order to navigate through a tree. For instance, if `t` corresponds to the subscript 1 in (3.1), then

```
(select t '(:up :next))
```

returns the list with one element "$+\cdots+a$". The routine `select` is implicitly called by many routines which operate on trees. For instance, with `t` as above,

```
(tree-ref t :up :next)
```

directly returns the tree "$+\cdots+a$".

Besides simpler access to subtrees of a tree or other "close trees", TEXMACS also provides several other useful mechanisms for writing editing routines. For instance, the routine `tree-innermost` and the macro `with-innermost` may be used to retrieve the innermost supertree of a certain type at the current cursor position. Since many editing routines operate at the current cursor position, two other useful macros are `with-cursor` and `cursor-after`, which allow you to perform some operations at a temporarily distinct cursor position resp. to compute the cursor position after some operations, without actually changing the current cursor position.

**A worked example.**

In order to illustrate the TEXMACS API for editing documents on a simple example, assume that we wish to write a function `swap-numerator-denominator` which allows us to swap the numerator and the denominator of the innermost fraction at the current cursor position.

The innermost fraction may simply be retrieved using the macro `with-innermost`. Together with the routine `tree-set!` for modifying a tree, this yields a first simple implementation:

```
(define (swap-numerator-denominator)
  (with-innermost t 'frac
    (tree-set! t `(frac ,(tree-ref t 1) ,(tree-ref t 0)))))
```

It should be noticed that the macro `with-innermost` ignores its body whenever no innermost fraction is found.

The above implementation has the disadvantage that we loose the current cursor position inside the numerator or denominator (wherever we were). The following refined implementation allows us to remain at the "same position" modulo the exchange numerator/denominator:

```
(define (swap-numerator-denominator)
  (with-innermost t 'frac
    (with p (tree-cursor-path t)
      (tree-set! t `(frac ,(tree-ref t 1) ,(tree-ref t 0)))
      (tree-go-to t (cons (- 1 (car p)) (cdr p))))))
```

Here we used the routines `tree-cursor-path` and `tree-go-to`, which allow us to manipulate the cursor position relative to a given tree.

As the icing on the cake, we may make our routine available through the mechanism of structured variants:

```
(define (variant-circulate t forward?)
  (:require (tree-is? t 'frac))
  (swap-numerator-denominator))
```

Notice that this implementation can be incorrect when operating on nested fractions. The implementation can be further improved by letting `swap-numerator-denominator` operate on a specific tree:

```
(define (swap-numerator-denominator t)
  (:require (tree-is? t 'frac))
  (with p (tree-cursor-path t)
    (tree-set! t `(frac ,(tree-ref t 1) ,(tree-ref t 0)))
    (tree-go-to t (cons (- 1 (car p)) (cdr p)))))
```

The corresponding generic routine could be defined as

```
(define (swap-numerator-denominator t)
  (and-with p (tree-outer t)
    (swap-numerator-denominator p)))
```

This piece of code will perform an outward recursion until a specific handler is found. We may now replace the call `(swap-numerator-denominator)` by `(swap-numerator-denominator (cursor-tree))`.

The new implementation also allows us to toggle the numerator and denominator of a selected fraction using `(swap-numerator-denominator (focus-tree))`. However, the focus is not necessarily conserved during the operation, thereby disallowing to restore the original state by toggling a second time. We may explicitly conserve the focus as follows:

```
(define (swap-numerator-denominator t)
  (:require (tree-is? t 'frac))
  (with p (tree-cursor-path t)
    (tree-set! t `(frac ,(tree-ref t 1) ,(tree-ref t 0)))
    (tree-go-to t (cons (- 1 (car p)) (cdr p)))
    (tree-focus t)))
```

This routine will even work when we are inside a nested fraction and operating on the outer fraction.

## 3.2. Fundamental tree modification routines

From an internal point of view, all modifications to the TeXmacs edit tree are decomposed into atomic modifications of eight different types. In this section, we describe the Scheme interface to these fundamental modification routines. Even though it is usually more convenient to use higher level modification routines, as described in the next section, the fundamental tree modification routines may occasionally be useful as well.

It should be emphasized that the fundamental tree modification routines are *not* checked for their correctness. It is the responsibility of the user to verify that the operations are valid and that they lead to a correct new edit tree. Although it is sometimes possible to leave the edit tree in a temporarily incorrect or "unsimplified" state (for instance, by allowing subtrees of the form `(concat "")`), this practice is not generally recommended, and may lead to severe bugs.

`(tree-assign! var new-value)` (tree assignment)

On input, we have a Scheme variable `var` of type `tree` and `new-value` of type `content`. The macro replaces the tree by `new-value` and updates `var` accordingly. The new tree value of `var` is returned.

`(tree-insert! var pos ins)` (insertion of new nodes or characters)

The first parameter `var` is a Scheme variable of type `tree`. If `var` is a compound tree, then `ins` should be a list $u_0, ..., u_{l-1}$ of new children of type `content`. In that case, the routine inserts $u_0, ..., u_{l-1}$ into the children of `var`, at position `pos` (see figure 3.1). If `var` is a string tree, then `ins` should be of string content type, and the string `ins` is inserted into `var` at position `pos`. The variable `var` is updated with the result of the insertion and the result is returned.

`(tree-remove! var pos nr)` (removal of nodes or characters)

The first parameter `var` is a Scheme variable of type `tree`. If `var` is a compound tree, then `nr` of its children are removed, starting at position `pos` (see figure 3.1). If `var` is a string tree, then `nr` characters are removed, starting at position `pos`. The variable `var` is updated with the result of the removal and the result is returned.

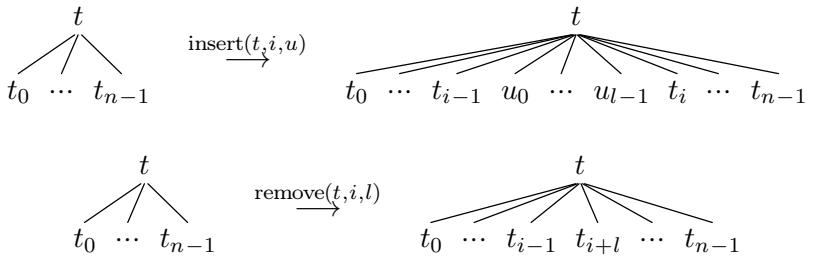



**Figure 3.1.** Illustration of the operations `(tree-insert! t i u)` and `(tree-remove! t i l)`. If `u` has length `l`, then we notice that `(tree-remove! t i l)` undos the insertion `(tree-insert! t i u)`.

`(tree-split! var pos at)` (split the children into two parts)

The first parameter `var` is a SCHEME variable of type `tree`. The macro is used to split the child $u$ of `var` at position `pos` into two parts. If $u$ is a compound tree, then the first part consists of the first `at` children and the second part of the remaining ones. Both parts carry the same label as $u$ and $u$ is replaced by the two parts inside `var` (see figure 3.2). If $u$ is string tree, then it is rather split into two strings at position `at`. The variable `var` is updated with the result of the split command and the result is returned.

`(tree-join! var pos)` (join two adjacent nodes)

The first parameter `var` is a SCHEME variable of type `tree`. This macro is used to join the child $u$ of `var` at position `pos` with the child $v$ at position `pos`+1. If $u$ and $v$ are trees, then they are removed from `var` and replaced by a single tree which has the same label as $u$ and whose children are those of $u$, followed by the children of $v$ (see figure 3.2). If $u$ and $v$ are strings, then they are replaced by their concatenation. The variable `var` is updated with the result of the join command and the result is returned.

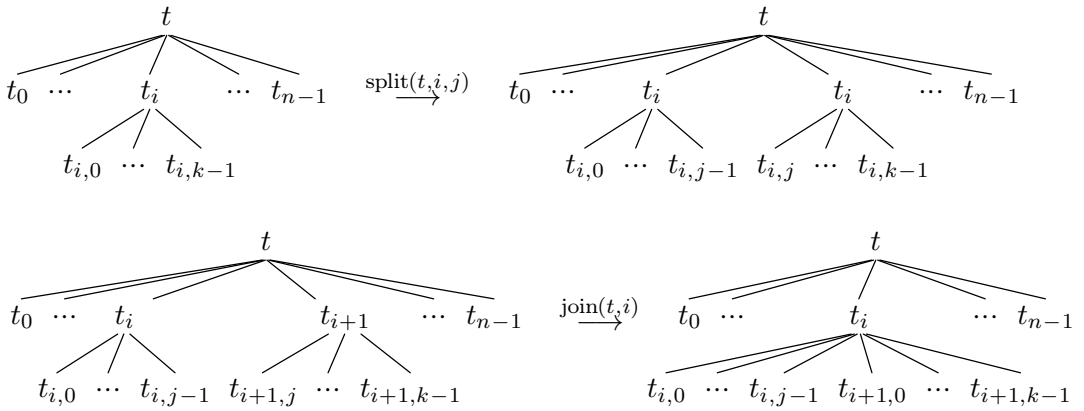



**Figure 3.2.** Illustration of the operations `(tree-split! t i j)` and `(tree-join! t i)`. Notice that `(tree-join! t i)` undos `(tree-split! t i j)`.

`(tree-assign-node! var lab)` (assign the label of a tree)

This macro replaces the label of a compound tree stored in a SCHEME variable `var` by a new value `lab`. The result of the substitution is returned.

`(tree-insert-node! var pos ins)` (insert the tree as a child of another one)

Given a SCHEME variable `var`, containing a tree, and a content tree `ins`, this macro replaces `var` by `ins`, with `var` inserted as a new child of `ins` at position `pos` (see figure 3.3). The result of the insertion is returned.

`(tree-remove-node! var pos)` (replace a tree by a child)

Given a Scheme variable `var`, containing a compound tree, this macro replaces `var` by its child at position `pos` (see figure 3.3). The value of this child is returned.

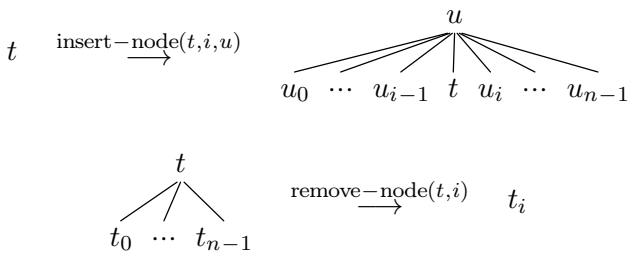



**Figure 3.3.** Illustration of the operations `(tree-insert-node! t i u)` and `(tree-remove-node! t i)`. Notice that the second operation undoes the first one.

**Remark 3.1.** Each of the macros `tree-assign!`, `tree-insert!`, etc. has a functional counterpart `tree-assign`, `tree-insert`, etc. The first parameter of these counterparts can be an arbitrary "l-value" and does not have to be a scheme variable. However, in the case when a Scheme variable is passed as the first parameter, these variants do not necessarily update its contents with the return value.

## 3.3. High level modification routines

The routine `tree-set` and the corresponding macro `tree-set!` can be used as a higher level interface to the fundamental routines for modifying trees as described in the previous section. However, it is still up to the user to verify that the resulting edit tree is still correct.

`(tree-set which accessors* new-value)`
`(tree-set! which accessors* new-value)` (smart tree assignment)

This routine replaces the tree `(tree-ref which accessors*)` by a new content value `new-value`. Besides the fact that the routine `tree-set` supports additional accessors for `which` (see the description of `tree-ref` below), `tree-set` differs from `tree-assign` in this respect that `tree-set` tries to cleverly decompose the assignment into fundamental modification routines. The objective of this decomposition is to make a less intrusive modifications in the document, so as to preserve as many tree positions and cursor positions as possible.

For instance, the operation `(tree-set t t)` is a no-operation for all trees `t`. A more complex operations like

```
(tree-set! t `(foo "Hop" ,(tree-ref t 2)))
```

is decomposed into the following fundamental modifications:

```
(tree-remove-node! t 2)
(tree-insert-node! t 1 '(foo "Hop"))
```

Like in the case `tree-assign` and `tree-assign!`, you should use the macro `tree-set!` in order to update the value of `which` if `which` is a Scheme variable `accessors` is the empty list.

`(tree-ref which accessors*)` (enhanced tree access)

In its simplest form, this routine allows for the quick access of a subtree of `which` via a list of integers `accessors`. For instance, if `which` contains the tree `(frac "a" (sqrt "b"))`, then `(tree-ref which 1 0)` returns the tree `"b"`.

In its general form, `tree-ref` relies on the routine `select` in order to compute the desired subtree. With `which` as in the above example, this makes it possible to retrieve the subtree `(sqrt "b")` using `(tree-ref t 'frac)`. In the case when there are several matches, the first match is returned. For instance, if `which` contains the tree `(frac (sqrt "a") (sqrt "b"))`, then `(tree-ref t 'frac)` returns `(sqrt "a")`.

In fact, the result of `tree-ref` is not necessarily a subtree of `which`: the `select` utility also accepts the accessors `:up`, `:down`, `:next`, `:previous`, etc. for navigating inside the edit tree starting with `which`. For instance, `(tree-ref (cursor-tree) :up)` returns the parent of the cursor tree. For more details, we refer to the documentation of `select`.

Besides the above routine for the direct modification of a subtree of the document, TEXMACS also provides several routines for inserting content at the current cursor position.

`(insert what)` (insertion of content)

Insert the content `what` at the current cursor position. TEXMACS does some additional checking whether it is allowed to perform the insertion. For instance, it is disallowed to insert multi-paragraph content inside a mathematical formula. Whenever the user attempts to make an invalid insertion, then `insert` is equivalent to a no-operation.

`(make lab)` (insertion of a tag)

This routine may be used to insert a valid tag with label `lab`. As many empty arguments as necessary are inserted in order to make the tag valid. Similarly, if `lab` is a multi-paragraph tag, then the necessary operations are performed to put the tag in a separate paragraph.

`make-with`, `insert-return`, etc.

## 3.4. PATH-BASED NAVIGATION

# Chapter 4

# TEXMACS buffer management

## 4.1. Introduction

There are three main kinds of objects for buffer management in TEXMACS:

**Buffers.** Every open TEXMACS document is stored in a unique editable buffer. Buffers typically admit a one to one correspondence to files on disk or elsewhere on the web. Some buffers are of a more auxiliary nature, such as automatically generated help buffers. All buffers admit a unique URL. In the case of auxiliary buffers, this URL is really a read-only "placeholder", so saving this kind of buffers is impossible (of course, it remains possible to save the buffer under a new name).

**Views.** It is possible to have multiple views on the same buffer. Every view is identified by a unique automatically generated URL, which again acts as a placeholder.

**Windows.** Views (contrary to the buffers themselves) can be displayed in actual windows. Currently, any TEXMACS window contains a unique view and a view may only be displayed in one window at the same time (of course, it is possible to display different views on the same buffer in different windows). Windows are again represented by automatically generated URLS.

**Remark 4.1.** In the future, views and windows should really be considered as documents themselves. Changes in the view will be automatically propagated (or not) to the corresponding buffer, and the other views. Windows will contain a document which specifies its layout (menus and toolbars). The corresponding view (or views) will be an active hyperlink (or active hyperlinks). The current APIs already reflect these future development intentions.

## 4.2. Manipulating TEXMACS buffers

**Basic buffer management.**

`(buffer-list)` (list of all buffers)

This routine returns the list of all open buffers.

`(current-buffer)` (current buffer)

Return the current view. The program may abort if there exists no current buffer.

`(path->buffer p)` (buffer which contains a certain path)

Return the buffer which contains a certain path `p`, or `#f`.

`(tree->buffer t)` (buffer which contains a certain tree)

Return the buffer which contains a certain tree `t`, or `#f`.

`(buffer->views buf)` (list of views on a buffer)

This routine returns the list of views on the buffer `buf`.

`(buffer->windows buf)` (list of windows containing buffer)

This routine returns the list of windows in which the buffer `buf` is currently being displayed.

`(buffer-new)` (create a new buffer)

Create a new buffer and returns its URL.

`(buffer-rename buf new-name)` (create a new buffer)

Give a new name `new-name` to the buffer `buf`.

`(switch-to-buffer buf)` (switch the editor's focus)

Switch the editor's focus to the buffer `buf`.

**Information associated to buffers.**

`(buffer-set buf rich-t)`
`(buffer-get buf)` (set/get the contents of the buffer)

Set the contents of the buffer `buf` to the rich tree `rich-t`, resp. get the rich contents of `buf`. Rich trees do not only contain the actual body of the document, but also some meta-data, such as its style, initial values of environment variables, and other auxiliary data attached to the document.

`(buffer-set-body buf t)`
`(buffer-get-body buf)` (set/get the main body of the buffer)

Set the main body of the buffer `buf` to the tree `t`, resp. get the main body of `buf`.

`(buffer-set-master buf master)`
`(buffer-get-master buf)` (set/get the master of the buffer)

Set the master of the buffer `buf` to `master`, resp. get the master of `buf`. The master of a buffer should again be a buffer. Usually, the master of a buffer is the buffer itself. Otherwise, the buffer will behave similarly as its master in some respects. For instance, if a buffer `a/b.tm` admits `x/y.tm` as its master, then a hyperlink to `c.tm` will point to `x/c.tm` and not to `a/c.tm`.

`(buffer-set-title buf name)`
`(buffer-get-title buf)` (set/get the title of the buffer)

Set the title of the buffer `buf` to the string `name`, resp. get the title of `buf`. The title is for instance used as the title for the window.

`(buffer-set-title buf name)`
`(buffer-get-title buf)` (set/get the title of the buffer)

Set the title of the buffer `buf` to the string `name`, resp. get the title of `buf`. The title is for instance used as the title for the window.

(buffer-last-save buf)
(buffer-last-visited buf) (time when a buffer was visited/saved last)

Return the time when the buffer buf was visited or saved last.

(buffer-modified? buf)
(buffer-pretend-saved buf) (check for modifications since last save)

The predicate buffer-modified? check whether the buffer buf was modified since the last time it was saved. The routine buffer-pretend-saved can be used in order to pretend that the buffer buf was saved, without actually saving it. This can for instance be useful if no worthwhile changes occurred in the buffer since the genuine last save.

**Synchronizing with the external world.**

Buffers inside TeXmacs usually correspond to actual files on disk or elsewhere. When changes occur on either side (e.g. when editing the buffer, or modifying the file on disk using an external program), the following routines can be used in order to synchronize the buffer inside TeXmacs with its corresponding file on disk.

(buffer-load buf) (load buffer)

Retrieve the buffer buf from disk (or elsewhere). Returns #t on error and #f otherwise. The format being used for loading files is chosen as a function of the extension of buf.

(buffer-save buf) (save buffer)

Save the buffer buf to disk (or elsewhere). Returns #t on error and #f otherwise. The format being used for saving files is chosen as a function of the extension of buf.

(buffer-import buf src fm) (import buffer)

Import the buffer buf from src, using the format fm. Returns #t on error and #f otherwise.

(buffer-export buf dest fm) (export buffer)

Export the buffer buf to dest, using the format fm. Returns #t on error and #f otherwise.

(tree-import src fm) (import a tree)

Import a tree from the URL src, using the format fm.

(tree-export t dest fm) (export a tree)

Export a tree to the URL dest, using the format fm.

## 4.3. Manipulating TeXmacs views

(view-list) (list of all views)

This routine returns the list of all available views, sorted by inverse chronological order. That is, views which were selected more recently will occur earlier in the list.

(current-view) (current view)

Return the current view or #f.

`(view->buffer vw)` (buffer to which the view is attached)

This routine returns the buffer to which the view `vw` is attached.

`(view->window vw)` (window to which the view is attached)

This routine returns the window in which the view `vw` is being displayed or `#f`.

`(view-new buf)`
`(view-passive buf)`
`(view-recent buf)` (get view on buffer)

All three routines return a view on the buffer `buf`. In the case of `view-new`, we systematically create a new view. The routine `view-passive` first attempts to find an existing view on `buf` which is not attached to a window; if no such view exists, then a new one is created. The last routine `view-recent` returns the most recent existing view, with a preference for the current view, or another visible view. Again, a new view is created if no suitable recent view exists.

## 4.4. MANIPULATING TEXMACS WINDOWS

`(window-list)` (list of all TEXMACS windows)

Return the list of all TEXMACS windows.

`(current-window)` (current window)

Return the current window. The program may abort if there exists no current window.

`(window->buffer win)` (buffer displayed in window)

This routine returns the buffer which is currently being displayed in the window `win`. Warning: in the future, when a window will be allowed to contain multiple buffers, this routine might be replaced by `window->buffers`.

`(window->view win)` (view displayed in window)

This routine returns the view which is currently being displayed in the window `win`. Warning: in the future, when a window will be allowed to contain multiple views, this routine might be replaced by `window->views`.

`(window-set-buffer win buf)` (show buffer in window)

Display the buffer `buf` in the window `win`.

`(window-set-view win vw)` (show view in window)

Display the view `vw` in the window `win`. The program may abort if the view was already attached to another window.

`(window-focus win)` (focus window)

Set the current focus to the window `win`. The current implementation is still a bit bugged and only correct if you want to execute a sequence of commands under the assumption that `win` carries the focus and if you return the focus to the original window at the end.

`(open-window)` (create new window)

Create a new window with an empty buffer and return the URL of the window.

`(open-buffer-in-window buf cnt attrs)` (No synopsis available)

Create a new window and set its main buffer to that identified by the URL `buf`. If `buf` is not yet a valid buffer, it is created and its contents set to `cnt`, otherwise the second parameter is ignored. The window is created with its attributes set to `attrs` (currently only the geometry is taken into account, but this might be extended in the future, see the C++ function `url new_window (bool map_flag= true, tree geom= "")`)

# Chapter 5

# Scheme interface for the graphical mode

## 5.1. Low level graphics manipulation

**Rationale.**

TEXMACS provides a small low-level library for the manipulation of graphics on top of the usual tree interface. One particularity of graphics operations is that they usually concern a large number of continuous changes (as a function of mouse movement) to one or more objects (under construction or being edited). On the one hand, this means that not all movements have to be undoable. On the other hand, this implies that some optimizations may be necessary to obtain a reasonable speed.

For these reasons, the library allows the programmer to focus attention on one or several objects in a graphics and to quickly perform operations on these objects. Focus is mostly understood to be temporary: typically, the focus is released as soon as an operation has been completed, i.e. the construction of a polyline.

From the implementation point of view, the selected objects may either be removed from the document (current implementation), or kept in the document (future implementation), while displaying them on top of the other objects (if necessary).

**Definitions.**

**Tree.** As in the main tree API. There are three main types of trees with graphical markup: graphics, shapes and groups.

**Enhanced tree.** Trees with graphical markup can be enhanced to provide additional properties for the markup by means of with tags. For instance, an "enhanced shape" (see below) might be a polyline together with a particular color and line width.

**Radical and properties.** In the case of an enhanced tree of the form `(with props* object)`, `object` is called the radical of the enhanced tree and `props*` the properties of the enhanced tree. Notice that an enhanced tree is allowed to be reduced to its radical, in which case it has no properties.

**Graphics.** This term corresponds to the main graphics, which is an ordered list of enhanced shapes or groups. Enhancements for the main graphics can be divided in two categories:

- Global properties for the graphics itself, e.g. rendering properties, or a background grid.
- Editing properties, which control the current editing behaviour of the graphics (polyline mode, current pen colour, etc.).

**Shape.** A shape is an atomic graphical markup primitive, such as a polyline. Typical enhancements for shapes are pen color, fill color, line width, arrow mode, etc.

**Group.** A group is an ordered list of enhanced shapes or groups. The possible enhancements for groups are the same as the ones for shapes (and, in this respect, groups therefore differ from graphics).

**Sketch.** The current sketch corresponds to a single or ordered list of enhanced shapes or groups on which the graphical editor is currently operating. There are two main modes for the sketch:

**SELECTING.** the sketch corresponds to a selection of enhanced shapes or groups in the main document.

**MODIFYING.** the sketch corresponds to a single or ordered list of enhanced shapes or groups which are being constructed or modified. The trees in the sketch can be new trees or trees which correspond to marked (invisible) trees in the main document.

The current sketch is usually displayed on top of all other graphics, together with several control points.

**Manipulation of enhanced trees.**

`(enhanced-tree->radical t)` (get radical)

Given an enhanced tree `t`, return its radical.

`(radical->enhanced-tree t)` (get enhanced tree)

Given a radical `t`, find its parent which corresponds to its largest enhancement. If `t` does not belong to a TeXmacs document, this routine returns `#f`.

`(enhanced-tree-set! t p* u)`
`(enhanced-tree-ref t p*)`
`(enhanced-tree-arity t p*)` (analogue of basic tree API)

These routines are similar to tree-set, `tree-set!`, etc. except that they operate on the radical of the enhanced tree.

`(enhanced-tree-properties-set! t l)` (set properties)

Given an enhanced tree `t`, override its properties with the elements in the association list `l`.

`(enhanced-tree-properties-ref t)` (get properties)

Obtain an association list with all properties of the enhanced tree `t`.

`(enhanced-tree-property-set! t var val)` (set enhanced property)

Set the property `var` of an enhanced tree `t` to `val`.

`(enhanced-tree-property-ref t var)` (get enhanced property)

Obtain the property `var` of an enhanced tree `t`.

**Sketch manipulation.**

`(sketch-tree)` (get current sketch)

Return the current sketch tree.

`(sketch-new t)` (start sketch)

Put a new tree in the sketch, which is not part of the document. This routine is typically called when starting the construction of a new enhanced shape.

`(sketch-set t)` (set sketch tree)

Assign the sketch which a tree `t` which is part of the document (and maintain the correspondence between `t` and the sketch). This routine is typically called when editing an enhanced shape.

`(sketch-reset)` (reset sketch tree)

Assign the sketch with an empty group of objects. This routine is typically called before starting the selection of a group of objects.

`(sketch-toggle t)` (toggle a tree in the sketch)

When the sketch is an enhanced group, this routine toggles whether a tree t in the document belongs to the group (and we maintain the correspondence between t and the corresponding subtree in the sketch). This routine is typically called when selecting a group of objects.

`(sketch-checkout)` (checkout the sketch)

Enter MODIFYING mode and potentially disable the counterparts of the trees in the sketch in the main document.

`(sketch-commit)` (commit the sketch)

Commit changes made to the sketch in MODIFYING mode and return to SELECTING mode.

`(sketch-cancel)` (cancel the sketch)

Cancel any changes made to the sketch in MODIFYING mode and return to the state of the document before the call of sketch-checkout.

**Miscellaneous.**

`(sketch-controls-set l)` (set controls)

Assign a list of markup objects with control ornaments to the current sketch. The ornaments are rendered on top of the sketch as a visual aid for the user. Typically, when editing a polyline, `l` consists of a list of control points.

## 5.2. Graphics interface between C++ and Scheme

**Rationale.**

TEXMACS both implements a low-level part of the graphics in C++ and the high-level user interface in Scheme. This API describes how both parts interact.

The low-level C++ mainly takes care of transforming the graphical markup in a typeset box. It also provides routines for translating between physical coordinates (relative to the window) into logical coordinates (the local coordinate system of the graphics) and routines for interacting with the typeset boxes (finding the closest objects to a given point or region or projecting a point on a grid).

**Definitions.**

**Editor coordinates.** The coordinates of the outermost typeset box. Mouse events are typically passed in these coordinates. The corresponding data type is `SI`.

**Graphics coordinates.** The coordinates of the innermost graphics corresponding to the current cursor position.

**Grid.** The current grid relative to the graphics for editing objects (this grid may theoretically be different from the grid which is displayed). The current grid consists both of a mathematical type of grid (no grid, cartesian grid, polar grid, etc.), together with special points which correspond either to control points, intersections of curves with the grid, intersections of curves, or self-intersections of curves.

**Grid point.** A point on the grid is a triple `(p distance type)`, where `p` is a point in graphics coordinates, `distance` its distance to the point which was projected on the grid (see `grid-project` below) and `type` the type of grid point with a potential origin. For instance, `type` can be `plain` or something like `(control t)` for a control point corresponding to the tree `t` in the document.

**Coordinate transformations.**

`(editor->graphics p)` (get graphics coordinates)

Transform a point `p` of the form `(x y)` from the editor coordinates into the graphics coordinates.

`(graphics->editor p)` (get editor coordinates)

Transform a point `p` of the form `(x y)` from the graphics coordinates into the editor coordinates.

**Grid routines.**

`(grid-project p)` (project point on grid)

Given a point `p` (in graphics coordinates), find its projection on the current grid, the `distance` part of the projection being the distance between `p` and its projection.

Note: the routine grid-project can also be used in order to find editable shapes and groups close to the current pointer position. Indeed, the corresponding control points are understood to lie on the grid in our sense.

`(grid-point-pertinence<? p q)`
`(grid-point-pertinence<=? p q)` (order by pertinence)

Grid points are ordered by pertinence as a function of type and distance. For instance, control points have higher pertinence than plain grid points and closer grid points are considered better than farther ones.

**Selection of shapes.**

`(graphics-find-disk p r)` (search shapes in disk)

Return the list of all trees in the graphics which intersect a disk with center `p` and radius `r` (in graphics coordinates).

`(graphics-find-rectangle p q)` (search shapes in rectangle)

Return the list of all trees in the graphics which intersect a rectangle with corners `p` and `q` (in graphics coordinates).

**Computations with shapes.**

`(box-info t)` (get bounding box for a shape)

Get a bounding box (and other information) about a shape `t`. `t` can be a tree or a scheme tree.

**Remark 5.1.** This section might be extended, since a lot of the graphical intelligence is implemented in the C++ code. For instance, we might want to compute the intersections of two curves inside the Scheme code. Also, when we will allow for user macros, we might want routines which return the graphical expansion of the macro (the constituent elementary shapes, i.e. polylines, splines, etc.).

# Chapter 6

# Extending the graphical user interface

Most of the user interface to TEXMACS is dynamically created from within the interpreted SCHEME code. New menus and buttons can be added, or the existing ones reused and rearranged, even the main editor can be embedded anywhere.

Imagine you want to implement some feature which requires interaction with the user. One possible approach is to use the facility `interactive`, which according to the user's preferences will either popup a dialog or ask in the footer bar, based in metadata you provide inside your `tm-define`'d function. See "Meta information and logical programming" for more on this topic. However, automatically generated content is not always the best approach, so you might want to explicitly design your interface placing it inside a complicated dialog. The following sections should help you with this.

## 6.1. An introduction to widgets

In TEXMACS you create and extend the visual interface using *widgets*. This word means either the basic building blocks you have at your disposal, like buttons, popup lists, etc. or the collections of those into dialogs, menu bars or whatever. This rather loose concept might be confusing, especially when we refer to what usually are know as dialogs as widgets, but it makes sense because all sorts of widgets can be aggregated to build more complicated ones as well.[6.1]

However, it must be kept in mind that items intended to be inserted in a menu bar won't necessarily display as they do in a separate window: complicated aggregations of widgets might be better placed in a separate window or dialogue, as explained in "Dialogs and composite widgets".

A complete reference with all the available widgets is the "Widgets reference guide", and you can find some examples in the other subsections of "Extending the graphical user interface". If you'd rather see the sources, the whole list of keywords is in the table `gui-make-table` inside `menu-define.scm`.

To create a widget, you'll first need to use `tm-widget` to define a new one. The call to this function uses its particular syntax, with many keywords for the creation of widgets. But we'll start with some buttons and labels.

Execute the following two lines to get the unavoidable example and leave your mouse over the "Hello" button.

```
Scheme] (tm-widget (example1) ("Hello" "world!"))
Scheme] (top-window example1 "A first try")
```

As you can see, buttons are implicitly created by simply writing a list with the button's title and a tooltip to be displayed when the user hovers over the button. A bit confusing, and also ugly, because this is intended for *toolbar* buttons. What you probably want is this:

6.1. If you miss some particular "building block" from your OS, you might see whether it's feasible as an aggregation of simpler ones or try and play with the UI interface code in C++ (but you'll have to add it for every supported platform!).

```
Scheme] (tm-widget (example2) (explicit-buttons ("Hello" (noop))))
Scheme] (top-window example2 "A nicer button")
```

The second argument is now a SCHEME command to be executed when the user clicks the button, in this case a no-operation, or `(noop)`. Try changing it for `(display "World")` or anything that suits you.

The next step is to add some text next to the button, i.e. a label. This is done with the `text` keyword, as in `(text "Hello")`, but in order to have both widgets sit side by side, you'll need a container widget as described in "Containers, glue, refresh and co.", such as `hlist`:

```
Scheme] (tm-widget (example3)
          (hlist
            (text "Hello")
            (explicit-buttons ("world" (display "!\n")))))

Scheme] (top-window example3 "Some text")
```

That was nice, but as you see, the two widgets are packed together until you resize the window. We need to explicitly tell TEXMACS to insert some space between them:

```
Scheme] (tm-widget (example3)
          (hlist
            (text "Hello")
            >>>
            (explicit-buttons ("world" (display "!\n")))))

Scheme] (top-window example3 "Some text")
```

The special symbol `>>>` is just one of the predefined glue widgets described in "Containers, glue, refresh and co.".

Text attributes may be changed for `text` widgets and many others by enclosing them inside what we'll name *style widgets*. These attributes are `mini`, `monospaced`, `grey`, `inert`, `centered` and `bold`, and respectively: reduce the size of the widget, choose a `monospaced` `font`, set the color to grey, deactivate the widget (meaning it is rendered, but greyed out and inactive), center it and choose a bold face. Here is an example:

```
Scheme] (tm-widget (example3)
          (hlist
            (bold (text "Hello"))
            >>>
            (inert (explicit-buttons ("world" (display "!\n"))))))

Scheme] (top-window example3 "Some text")
```

From here you can go on reading "Extending the graphical user interface" or see the sample widgets in `menu-test.scm`.

## 6.2. MENUS AND TOOLBARS

As we said before, menus are special collections of widgets:

Problems with toolbars, system menus, context menus... Menu containers: horizontal menu, vertical menu. Separators.

## 6.3. Displaying lists and trees

### Displaying lists with **enum**, **choice** and **choices**.

`(enum cmd items default width)` (a combo box)

Builds a combo box which will execute `cmd` whenever the user makes a choice. The `width` may be given in any TeXmacs length unit.

```
Scheme] (tm-widget (test-enum)
          (enum (display* "First " answer "\n")
                '("gnu" "gnat" "zebra")
                "zebra" "10em")))

Scheme] (show test-enum)
```

`(choice cmd items default)` (a list of items allowing one to be chosen)

Builds a list of items which will execute `cmd` whenever the user makes a choice. `items` is a list, `default` a value. Contrary to `enum`, all items are displayed simultaneously. If one desires scrollbars, the widget must be enclosed in a `scrollable` container. The width of the widget may be set using a `resize` widget.

```
Scheme] (tm-widget (test-choice)
          (resize "200px" "50px"
            (scrollable
              (choice (display* answer "\n")
                      '("First" "Second" "Third" "Fourth" "Fifth"
        "Sixth")
                      "Third"))))
Scheme] (show test-choice)
```

`(choices cmd items defaults)` (a list of items allowing several to be chosen)

Builds a list of items which will execute `cmd` whenever the user makes a choice. Several items may be selected at the same time. Both `items` and `defaults` are hence lists. Contrary to `enum`, all items are displayed simultaneously. If one desires scrollbars, the widget must be enclosed in a `scrollable` container. The width of the widget may be set using a `resize` widget.

```
Scheme] (tm-widget (test-choices)
          (resize "200px" "100px"
            (scrollable
              (choices (display* answer "\n")
                       '("A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L")
                       '("B" "D" "F" "H" "J" "L")))))
Scheme] (show test-choices)
```

### Displaying trees with **tree-widget**.

`(tree-widget cmd data data-roles)` (a tree view)

The `tree-widget` provides a graphical representation of a TeXmacs tree `data` (not a Scheme tree!). This may be part of a document or any other tree. The first node in `data` won't be displayed. All other nodes may have attributes called *data roles* which will determine the textual representation of the node, whether it has some icon next to it and which one, etc. These attributes are simply children of the nodes in `data` at predefined positions given by the data roles specification in the argument `data-roles`. This is a list of identifiers for each tree label present in the data. For instance, with the following data roles specification:

```
(list
  (library    DisplayRole DecorationRole UserRole:1)
  (collection DisplayRole UserRole:1))
```

we use the data:

```
(root
  (library "Library" "icon.png" 12345
    (collection "Cool stuff" 001)
    (collection "Things to read" 002)
    (collection "Current work" 003
      (collection "Forever current" 004)
      (collection "Read me" 005))))
```

Notice that the node `root` won't be displayed by the `tree-widget` and needs no data roles. Here `UserRole:1` is used to store database ids but it can be anything else. The supported data roles are:

```
DisplayRole         ; a string to be displayed
EditRole            ; a string valid for an editable representation
ToolTipRole         ; a small tooltip to display when the mouse hovers
over
StatusTipRole       ; for the status bar (if present and supported)
DecorationRole      ; file name of an icon to use
CommandRole         ; sent to the command executed after (double?) clicks
UserRole:<number>   ; left to user definition (will be returned as
strings)
```

**Default data roles.** It is possible to omit some or all of the data role specification. By default the widget will use the tree label's string representation for `DisplayRole`, `EditRole`, `ToolTipRole` and `StatusTipRole`. For the `DecorationRole` it will try to load pixmaps named `treelabel-<label>.xpm` in `$TEXMACS_PIXMAP_PATH`. This search **won't** happen if the `DecorationRole` is specified (i.e. a full path with or without environment variables and wildcards must be given). The default `CommandRole` is the subtree itself (see below).

**Using commands.** The first argument of `tree-widget`, `cmd`, is a SCHEME lambda that will be called when items are clicked. The procedure must have the following signature:

```
(lambda (Event CommandRole . UserRoles) (...))
```

where:

- `Event` is an integer: either 1, 2 or 4 for a single, right or middle click respectively. In the future, other events could be supported (like double clicks, drag&drop, unfold, etc.)
- `CommandRole` is either the value of that role if given for the data item, or the subtree itself otherwise.
- `UserRoles` is a (possibly empty) list with the data for those roles given in the data role specification.

If multiple selections are enabled and one is made, `CommandRole` and `UserRole` will both be lists (not implemented yet). Keep in mind that the data is a TeXmacs tree and thus not a copy but always a pointer to the actual data (unless you copy or transform it into another format with e.g. `tree->stree`)

**Examples.** See `widget10` in `menu-test.scm` and "Displaying lists and trees".

**An example using data roles.**

We build on the previous example, but now we add a command. Notice how the way one adds commands to `tree-view` departs from that of other widgets, where instead of a procedure one must provide a list with code expecting one or two arguments with fixed names (usually `answer` and `filter`). *Note to self*: this is easily changed in `$tree-view`, but it seems easier to manage empty arguments this way.

```
Scheme] (define t
          (stree->tree
           '(root
             (library "Library" "$TEXMACS_PIXMAP_PATH/tm_german.xpm" 01
                      (collection "Cool stuff" 001)
                      (collection "Things to read" 002)
                      (collection "Current work" 003
                                  (collection "Forever current" 004)
                                  (collection "Read me" 005))))))
Scheme] (define dd
          (stree->tree
           '(list (library DisplayRole DecorationRole UserRole:1)
                  (collection DisplayRole UserRole:1))))
Scheme] (define (action clicked cmd-role . user-roles)
          (display* "clicked= " clicked ", cmd-role= " cmd-role
                    ", user-roles= " user-roles "\n")))
Scheme] (tm-widget (widget-library)
          (resize ("150px" "400px" "9000px") ("300px" "600px" "9000px")
            (vertical
              (bold (text "Testing tree-view"))
              ===
              (tree-view action t dd))))
Scheme] (top-window widget-library "Tree View")
```

Notice how we must add `$TEXMACS_PIXMAP_PATH` to the name of the pixmap because we are not using the default `DecorationRole`.

**An example using the buffer tree.**

We can even use the `buffer-tree` as argument to `tree-widget`. Changes in the buffer will show up immediately in the widget. In this example we use the default data role specification.

**Warning 6.1.** As of this writing (31 Dec. 2013) the Qt implementation is sloppy and forces a full reloading of the data model for each `modification` of the `tree`. The slowdown is already noticeable with documents of a few pages like this one. Additionally, the current selection in the widget is lost after each modification to the buffer (fixing this requires writing a fully fledged `observer` and probably an intermediate copy of the data).

```
Scheme] (tm-widget (widget-buffer)
          (resize ("150px" "400px" "9000px") ("300px" "600px" "9000px")
            (vertical
              (bold (text "Testing tree-view"))
              ===
              (tree-view noop (buffer-tree) (stree->tree '(dummy))))))
Scheme] (top-window widget-buffer "Tree View")
```

**An example with the side tools.**

If your TEXMACS has the side tools enabled, you can try this:

```
Scheme] (tm-widget (texmacs-side-tools)
          (vertical
            (hlist (glue #t #f 15 0) (text "Document tree:") (glue #t #f 15
        0))
            ---
            (tree-view noop (buffer-tree) (stree->tree '(unused)))))
```

## 6.4. DIALOGS AND COMPOSITE WIDGETS

*Dialogs* are collections of widgets arranged in a window in order to perform a common task. You might want to create one of this in order to configure or interact with a plugin: add some configuration options as well as some common actions and have the window always open besides your document. A good example whose code might help is the preferences dialog (`open-preferences`).

In order to create more complex layouts than those we did before you'll need a few containers. Among these are `aligned` and `tabs`, which we explain below. A very useful macro which you'll be using often is `dynamic`: it allows you to embed one widget into another.

Let's see how you create a dialog. To get started here is one little example taken from `menu-test.scm`:

```
Scheme] (tm-widget (widget1)
          (centered
            (aligned
              (item (text "First:")
                (toggle (display* "First " answer "\n") #f))
              (item (text "Second:")
                (toggle (display* "Second " answer "\n") #f)))))
```

The keyword `centered` is clear, just center whatever it contains, but `aligned` not so much: it builds two column tables, with each row of type `item`. As you can see, each `item` takes two arguments, which can be of *any* type.

The `toggle` is another example of a widget which triggers a SCHEME command whenever it's clicked, or toggled in this case. The second argument stands for the default state of the `toggle`.

Again, in order to display this you create a `top-window` and give it a title.

```
Scheme] (top-window widget1 "Two toggle widgets")
```

You'll notice that the created window is too small and the title is not wholly displayed. You can force it to be of a certain size using `resize`:

```
Scheme] (tm-widget (widget1)
          (centered
            (resize "500px" "200px"
              (aligned
                (item (text "First:")
                  (toggle (display* "First " answer "\n") #f))
                (item (text "Second:")
                  (toggle (display* "Second " answer "\n") #f))))))

Scheme] (top-window widget1 "A bigger window")
```

`resize` is another of the several available container or content management widgets. It accepts two sorts of arguments. Either one sets a fixed size for the widget with two strings, as in the example above, or one passes two lists, the first for widths, the second for heights, with the minimum, default and maximum values in that order, like this:

```
(resize ("100px" "200px" "400px") ("100px" "200px" "400px") (some-widget-
here))
```

This sets `some-widget-here` to have a default square size of 200x200 pixels.

If you want to add the usual buttons you use `bottom-buttons` like in the following example. Notice that the widget now accepts one parameter `cmd` which will be called when the user clicks the "Ok" button.

```
Scheme] (tm-widget (widget1-buttons cmd)
          (centered
            (aligned
              (item (text "First:")
                (toggle (display* "First " answer "\n") #f))
              (item (text "Second:")
                (toggle (display* "Second " answer "\n") #f))))
          (bottom-buttons >> ("Ok" (cmd "Ok"))))
```

Since the widget now needs an argument, we must use another function to display it, namely `dialogue-window`, which will also close the window after the button has been clicked.

```
Scheme] (dialogue-window widget1-buttons (lambda (arg) (display* arg "\n"))
        "Two toggles")
```

That special >> at the end of the widget inserts as before whitespace, but it stretches and aligns the `bottom-buttons` to the right. This is just another example of a glue widget.

### 6.4.1. Composite widgets

Note that our second dialog, `widget1-buttons` is just a copy of `widget1` with an extra line at the end. We could have spared us the keystrokes in this way:

```
Scheme] (tm-widget (widget1-buttons-smarter cmd)
          (dynamic (widget1))
          (bottom-buttons >> ("Ok" (cmd "Ok"))))

Scheme] (dialogue-window widget1-buttons-smarter (lambda (arg) (display* arg
        "\n")) "Two toggles")
```

As you can see, the approach we've shown has a shortcoming: there's no way to access all the values of the different widgets in your dialog at the same time. Of course you can use the function `cmd` passed to your widget to perform some computations, but in case you need to retrieve or store complicated data, what you need is a form.

## 6.5. Forms

As explained in "Dialogs and composite widgets" the available widgets can be used to compose dialog windows which perform one simple task. But sometimes one needs to read complex input from the user and forms provide one mechanism to do this. They allow you to define multiple named fields of several types, whose values are stored in a hash table. The contents of this hash can be retrieved when the user clicks a button using the functions `form-fields` and `form-values`.

In the following example you can see that the syntax is pretty much the same as for regular widgets, but you must prefix the keywords with `form-` :

```
Scheme] (tm-widget (form3 cmd)
          (resize "500px" "500px"
            (padded
              (form "Test"
                (aligned
                  (item (text "Input:")
                    (form-input "fieldname1" "string" '("one") "1w"))
                  (item === ===)
                  (item (text "Enum:")
                    (form-enum "fieldname2" '("one" "two" "three") "two"
        "1w"))
                  (item === ===)
                  (item (text "Choice:")
                    (form-choice "fieldname3" '("one" "two" "three") "one"))
                  (item === ===)
                  (item (text "Choices:")
                    (form-choices "fieldname4"
                                  '("one" "two" "three")
                                  '("one" "two"))))
                (bottom-buttons
                  ("Cancel" (cmd "cancel")) >>
                  ("Ok"
                   (display* (form-fields) " -> " (form-values) "\n")
                   (cmd "ok")))))))
Scheme] (dialogue-window form3 (lambda (x) (display* x "\n")) "Test of
        form3")
```

A complete list of the widgets you can embed in a form is in the table `gui-make-table` inside `menu-define.scm`.

## 6.6. Containers, glue, refresh and co.

### 6.6.1. Attribute widgets

In what follows `widget` can be anything defined using `tm-widget`.

`(centered widget)` (centers `widget`)

This does just that: it centers `widget` with respect to the enclosing widget. Although we are calling this an attribute, the effect is achieved by using a vertical list and a horizontal one together with four `glue` widgets. This means that in the following example, the first widget is actually expanded to something like the second one.

```
Scheme] (tm-widget (wid1)
          (centered (text "I'm centered.")))

   ((guile-user) (guile-user))

Scheme] (tm-widget (wid2)
          (vlist
            (glue #f #f 0 10)
            (hlist
              (glue #t #f 25 0)
              (text "I'm centered.")
              (glue #t #f 25 0))
            (glue #f #f 0 10)))
Scheme] (show wid1)
Scheme] (show wid2)
```

`(resize (w1 w2 w3) (h1 h2 h3) wid)`
`(resize w h widget)` (resizes `widget`)

These two variants resize the argument. The first one specifies a minimum size of `w1`×`h1`, a default size of `w2`×`h2` and a maximum size of `w3`×`h3`. `widget` will be set to the default size and will be allowed to resize but not beyond the bounds specified. The second alternative sets a fixed width and height.

Sizes are specified as strings with a unit suffix, like in `"150px"`.

```
Scheme] (tm-widget (wid)
          (resize "200px" "70px" (text "I'm stuck!")))


Scheme] (show wid)
```

`(padded widget)` (surrounds `widget` by padding)

This sets some fixed padding around `widget`. As in the case of `centered`, the effect is achieved by means of several widgets into which this macro expands. These are actually the same as in the example there, but the `glue` widgets are all fixed (i.e. have all their expansion parameters set to `#f`).

### 6.6.2. Container or layout widgets

You can arrange widgets horizontally or vertically, or in two column mode as in forms. When running the QT version the latter will default to the OS standard for arranging labels and their associated input widgets in dialogs. Other possibilities are splitters and tabbed widgets. A very useful macro is `dynamic`, which allows you to embed one widget into another.

`(aligned items-list)` (arranges items in a two column table)

`(hlist widgets)` (arranges items horizontally)

`(vlist widgets)` (arranges items vertically)

`(hsplit (item (widget)) (item (widget)) ...)` (arranges two items in a split panel)

`(tabs (tab (widget)) (tab (widget)) ...)` (a tabbed widget)

`(dynamic (widget))` (embeds a tm-widget into another one)

### 6.6.3. Glue widgets

Besides laying out widgets in containers, you will often want to specify how they eat up space around them when the user resizes the window. By default (most?) widgets take up as much space as they can (i.e. they always expand) unless you used `resize` with them or they can have their size set with a parameter. If you don't want this to happen you can place invisible spacers around them which will (if you tell them to) gobble up as much as they can, either vertically or horizontally or in both directions.

TEXMACS provides one such basic building block:

`(glue horiz vert width height)` (possibly expanding whitespace)

The first two parameters, `horiz` and `vert`, are of boolean type and specify whether the `glue` widget will try to expand horizontally or vertically when its surroundings do. The last two parameters, `width height`, either fix the size for non-expanding `glue` or set a minimum one.

```
Scheme] (tm-widget (wid1)
          (centered (text "I'm centered.")))

Scheme] (tm-widget (wid2)
          (vlist
            ===
            (hlist
              (glue #t #f 25 0)
              (text "I'm centered.")
              (glue #t #f 25 0))
            (glue #f #f 0 10)))

Scheme] (show wid1)
Scheme] (show wid2)
```

In addition to the basic `glue` widget, there are several convenience macros.

`===` (vertical separator)

Expands to `(glue #f #f 0 5)`.

`======` (big vertical separator)

Expands to `(glue #f #f 0 15)`.

`//` (horizontal separator)

Expands to `(glue #f #f 5 0)`.

`///` (big horizontal separator)

Expands to `(glue #f #f 15 0)`.

`>>` (expanding horizontal separator)

Expands to `(glue #t #f 5 0)`.

`>>>` (big expanding horizontal separator)

Expands to `(glue #t #f 15 0)`.

For the specific use in menus the following two macros are defined:

`|` (horizontal separator)

(A vertical bar)

`---` (vertical separator)

(Three dashes)

### 6.6.4. Refresh widgets

Refresh widgets redraw their contents every time a command is executed. They achieve this re-evaluating the code for the whole widget, so you can have new values in your variables...

## 6.7. Widgets reference guide

This should be a comprehensive list of all the widgets available to the user, following this schema:

`some-symbol` (Some synopsis)

Some explanation.

An excerpt from `progs/kernel/gui/menu-define.scm`, as of SVN revision 5238:

```
(define-table gui-make-table
  (eval ,gui-make-eval)
  (dynamic ,gui-make-dynamic)
  (former ,gui-make-former)
  (link ,gui-make-link)
  (let ,gui-make-let)
  (let* ,gui-make-let)
  (with ,gui-make-with)
```

```
(receive ,gui-make-with)
(for ,gui-make-for)
(cond ,gui-make-cond)
(refresh ,gui-make-refresh)
(group ,gui-make-group)
(text ,gui-make-text)
(glue ,gui-make-glue)
(color ,gui-make-color)
(texmacs-output ,gui-make-texmacs-output)
(texmacs-input ,gui-make-texmacs-input)
(input ,gui-make-input)
(enum ,gui-make-enum)
(choice ,gui-make-choice)
(choices ,gui-make-choices)
(toggle ,gui-make-toggle)
(icon ,gui-make-icon)
(concat ,gui-make-concat)
(verbatim ,gui-make-verbatim)
(check ,gui-make-check)
(balloon ,gui-make-balloon)
(-> ,gui-make-submenu)
(=> ,gui-make-top-submenu)
(horizontal ,gui-make-horizontal)
(vertical ,gui-make-vertical)
(hlist ,gui-make-hlist)
(vlist ,gui-make-vlist)
(aligned ,gui-make-aligned)
(item ,gui-make-item)
(meti ,gui-make-meti)
(tabs ,gui-make-tabs)
(tab ,gui-make-tab)
(inert ,gui-make-inert)
(explicit-buttons ,gui-make-explicit-buttons)
(bold ,gui-make-bold)
(tile ,gui-make-tile)
(scrollable ,gui-make-scrollable)
```

```
  (resize ,gui-make-resize)
  (hsplit ,gui-make-hsplit)
  (vsplit ,gui-make-vsplit)
  (minibar ,gui-make-minibar)
  (extend ,gui-make-extend)
  (padded ,gui-make-padded)
  (centered ,gui-make-centered)
  (bottom-buttons ,gui-make-bottom-buttons)
  (assuming ,gui-make-assuming)
  (if ,gui-make-if)
  (when ,gui-make-when)
  (mini ,gui-make-mini)
  (symbol ,gui-make-symbol)
  (promise ,gui-make-promise)
  (ink ,gui-make-ink)
  (form ,gui-make-form)
  (form-input ,gui-make-form-input)
  (form-enum ,gui-make-form-enum)
  (form-choice ,gui-make-form-choice)
  (form-choices ,gui-make-form-choices))

(tm-define (gui-make x)
  ;;(display* "x= " x "\n")
  (cond ((symbol? x)
         (cond ((== x '---) '$---)
               ((== x '===) (gui-make '(glue #f #f 0 5)))
               ((== x '======) (gui-make '(glue #f #f 0 15)))
               ((== x '/) '$/)
               ((== x '//) (gui-make '(glue #f #f 5 0)))
               ((== x '///) (gui-make '(glue #f #f 15 0)))
               ((== x '>>) (gui-make '(glue #t #f 5 0)))
               ((== x '>>>) (gui-make '(glue #t #f 15 0)))
               ((== x (string->symbol "|")) '$/)
               (else
                 (texmacs-error "gui-make" "invalid menu item ~S" x))))
        ((string? x) x)
```

```
((and (pair? x) (ahash-ref gui-make-table (car x)))
 (apply (car (ahash-ref gui-make-table (car x))) (list x)))
((and (pair? x) (or (string? (car x)) (pair? (car x))))
 `($> ,(gui-make (car x)) ,@(cdr x)))
(else
  (texmacs-error "gui-make" "invalid menu item ~S" x))))
```

# Chapter 7

# Writing TeXmacs bibliography styles

## 7.1. Introduction

TeXmacs admits support both for BibTeX and a native tool for managing bibliographies. BibTeX styles are denoted by their usual names. TeXmacs styles are prefixed by `tm-`. For example, the TeXmacs `tm-plain` style is the replacement for the BibTeX `plain` style. Equivalents for the following BibTeX styles have been implemented: `abbrv`, `alpha`, `ieeetr`, `plain` et `siam`. These styles can therefore be used without installation of BibTeX.

New bibliography styles can be defined by the user. Each style is associated to a unique Scheme file, which should be added to the directory `$TEXMACS_PATH/prog/bibtex`. Style files are treated as regular Scheme programs. Since the creation of a style file from scratch is a complex task, we recommend you customize existing style files or modules. In the next sections, we will describe the creation of a new style on a simple example and give a detailed lists of available Scheme functions which facilitate the creation of new styles.

## 7.2. Example of a simple bibliography style

Bibliographic style files are stored in directory `$TEXMACS_PATH/progs/bibtex`. They have the name of the style followed with extension `.scm`. For example, `example.scm` is the file name associated to the style `example`, which is denoted by `tm-example` when it is used in a TeXmacs document.

All style files must be declared as a module as follows:

```
(texmacs-module (bibtex example)
  (:use (bibtex bib-utils)))
```

The module `bib-utils` contains all useful functions needed to write bibliographic styles.

All style files must me declared as a bibliographic style as follows:

```
(bib-define-style "example" "plain")
```

The first argument to `bib-define-style` is the name of the current style. The second argument is the name of a fall-back style, `plain` in our case. If a function is not defined in current style, the function from the fall-back style is used instead. Hence, the following minimalistic style file behaves in an identical way as the `plain` style:

```
(texmacs-module (bibtex example)
  (:use (bibtex bib-utils)))

(bib-define-style "example" "plain")
```

Each formatting function defined in the default style can be overloaded in the current style. For example, the function `bib-format-date` is used to format the date in the `plain` style. It is redefinable in our example style as follows:

```
(tm-define (bib-format-date e)
  (:mode bib-example?)
  (bib-format-field e "year"))
```

All exported functions must be prefixed with `bib-`. Overloaded functions must be followed with directive `(:mode bib-example?)`, in which `example` is the name of the current style.

Our complete example file `example.scm` is as follows:

```
(texmacs-module (bibtex example)
  (:use (bibtex bib-utils)))

(bib-define-style "example" "plain")

(tm-define (bib-format-date e)
  (:mode bib-example?)
  (bib-format-field e "year"))
```

It behaves in a similar way as the `plain` style, except that all dates are formatted according to our custom routine.

## 7.3. SCHEME FUNCTIONS FOR WRITING BIBLIOGRAPHY STYLES

### 7.3.1. Style management

`(bib-define-style name default)` (style declaration)

This function declares a style called `name` (string) with fall-back style `default` (string). The style is selected by choosing `tm-name` when adding a bibliography to a document. Whenever a formatting function is not defined in the current style, its definition in the fall-back style is used as replacement.

`(bib-with-style style expr)` (local style)

This function evaluates expression `expr` as if the current style were `style` (string).

### 7.3.2. Field related routines

`(bib-field entry field)` (field data)

This function creates a TEXMACS tree corresponding to the field `field` (string) of entry `entry` without format. In some cases, the output is special:

- If `field` is `"author"` or `"editor"`, we return a tree with label `bib-names` followed by a list of author names; each author name is a tree with label `bib-name` containing four elements: first name, particule (von), last name and suffix (jr);
- If `field` is `"page"`, then we return a list of integers: the empty list, or a singleton with a page number, or a pair corresponding to a pages interval.

`(bib-format-field entry field)` (basic format)

This function creates a TEXMACS tree corresponding to the field `field` (string) of entry `entry`, with basic format.

`(bib-format-field-Locase entry field)` (special format)

This function is similar to `bib-format-field` ; but field are formatted in lower case with an upper case letter at the beginning.

`(bib-empty? entry field)` (null-test of a field)

This function returns boolean `#t` if the field `field` (string) of entry `entry` is empty or absent; it returns `#f` in the other cases.

### 7.3.3. Routines for structuring the output

`(bib-new-block tm)` (new block)

This function creates aTEXMACS tree consisting of a block containing TEXMACS tree `tm`.

`(bib-new-list sep ltm)` (separated list)

This function creates a TEXMACS tree which is the concatenation of all the elements of list `ltm` separated with TEXMACS tree `sep`.

`(bib-new-list-spc ltm)` (blank separated list)

This function is equivalent to the evaluation of `(bib-new-list " " ltm)`.

`(bib-new-sentence ltm)` (new sentence)

This function creates a TEXMACS tree corresponding to a sentence containing all the elements of list `ltm` separated by commas.

### 7.3.4. Routines for textual manipulations

`(bib-abbreviate name dot spc)` (name abbreviation)

This function creates a TEXMACS tree corresponding to the abbreviation of the name contained in `name` TEXMACS tree: it retrieves the list of first letters of each word, followed by `dot` (TEXMACS tree) and separated by `spc` (TEXMACS tree).

`(bib-add-period tm)` (dot)

This function creates a TEXMACS tree with a dot at the end of `tm`.

`(bib-default tm)` (default TEXMACS tree)

This function creates a TEXMACS tree without label `keep-case`.

`(bib-emphasize tm)` (italic)

This function creates a TEXMACS tree corresponding to the italic version of `tm`.

`(bib-locase tm)` (lower case)

This function creates a TEXMACS tree, which is equal to `tm` with all letters in lower case, except for those within `keep-case` blocks.

`(bib-prefix tm nbcar)` (beginning of a TEXMACS tree)

This function returns a string containing the first `nbcar` characters of `tm`.

`(bib-upcase tm)` (upper case)

This function creates a TEXMACS tree, which is equal to `tm` with all letters in upper case, except for those within `keep-case` blocks.

`(bib-upcase-first tm)` (upper case first letter)

This function creates a TEXMACS tree, which is equal to `tm` with its first letter in upper case, except inside `keep-case` blocks.

### 7.3.5. Miscellaneous routines

`(bib-null? v)` (null-test)

This function returns boolean `#t` if value `v` is empty; it returns `#f` in the other cases.

`(bib-purify tm)` (flattening of a TEXMACS tree)

This function returns a string made of all letters of the TEXMACS tree `tm`.

`(bib-simplify tm)` (simplification of a TEXMACS tree)

This function returns a TEXMACS tree corresponding to the simplification of TEXMACS tree `tm`.

`(bib-text-length tm)` (length of a TEXMACS tree)

This function returns the length of TEXMACS tree `tm`.

`(bib-translate msg)` (translation)

This function translates the string message `msg` from english into the current language.

# Chapter 8
# About the API documentation

Documentation for TEXMACS internal features and API is typically written as part of the general documentation, where it's most natural for someone reading the manual as a book. However it often happens that some SCHEME module or procedure needs documenting but doesn't fit into any of the available sections of the manual. The purpose of this section is precisely to assemble all those pieces of information. Currently (jan. 2016) there are very sketchy pages for:

## 8.1. The TEXMACS file system

The TEXMACS file system is a complicated beast, with versioning, network access and authentication built in among other things. This documentation should be completed with all those features, but in the meantime, we have the following:

### 8.1.1. A `tmfs` primer

### 8.1.2. The TEXMACS filesystem

Many things in TEXMACS can be referenced through a URI with `tmfs` as schema. Examples of entities in this system are buffers, views and windows or at a higher level help buffers and search results. A TEXMACS URI follows the format:

```
tmfs://handler[/query]
```

Requests to open URIs such as these are sent to a *handler*, which actually is a set of procedures implementing the basic operations related to the type of content they handle: loading the content, saving it (if possible or necessary), setting the window title and establishing access permissions are the basic operations. Predefined handlers which the user usually encounters are `grep`, `help`, `history`, `revision` and `apidoc`: they accept a query representing search strings, files or help pages and render results in the appropriate language into a new buffer. The *query* is a string in the usual format `variable1=value1&variable2=value2`. Its parsing can be done using `query-ref`.

Situations where using this system makes more sense than regular documents are for instance documentation, which must be chosen from several languages and possibly be compiled on the fly from various sources (see module `doc.apidoc` and related modules) and automatically generated content, like that resulting from interacting from an external system for version control of documents (see handler `version` in module `version.version-tmfs`).

### 8.1.3. Implementing a handler

The definition of a handler is done via `tmfs-handler` or with the convenience macros `tmfs-load-handler`, `tmfs-save-handler`, `tmfs-permission-handler` and `tmfs-title-handler`.

Below we'll implement a basic load handler named `simple` which will accept two sorts of arguments: `type` and `what`. We shall use two procedures, one to handle the requests, another to create the document.

```
Scheme] (tm-define (simple-load header body)
          `(document
             (TeXmacs ,(texmacs-version))
             (style (tuple "generic"))
             (body (document (section ,header) ,body))))
```

As you can see, we don't do much other than creating a TeXmacs document. The load handler won't be complicated either. We only parse the query string with the help of `query-ref` and then display one of three possible buffers.

```
Scheme] (tmfs-load-handler (simple qry)
          (let ((type (query-ref qry "type"))
                (what (query-ref qry "what")))
            (tm->stree
              (cond ((== type "very") (simple-load "Very simple" what))
                    ((== type "totally") (simple-load "Totally simple"
        what))
                    (else (simple-load "Error"
                                       (string-append "Query unknown: " what)))))))
```

We can test this right away with:

```
Scheme] (load-buffer "tmfs://simple/type=very&what=example")
```

Or embedded in a document using tags like hlink and branch: click here to test it.

You can set read/write permissions implementing a *permission handler*, and the window's title using a *title handler*:

```
Scheme] (tmfs-permission-handler (simple name type)
          (display* "Name= " name "\nType= " type "\n")
          #t)
Scheme] (tmfs-title-handler (simple qry doc) "Simple handler - Some title
        here")
```

`(tmfs-load-handler (name qry) body)` (define load handler for @name)

A *load handler* for `name` is invoked when TeXmacs receives a request to open a URI of type `tmfs://name/qry`. The `body` of the handler is passed `qry` as parameter (see `query-ref`) and must return a complete TeXmacs buffer. Consider the following example:

```
(tmfs-load-handler (id qry)
  `(document
     (TeXmacs ,(texmacs-version))
     (style (tuple "generic"))
     (body (document ,qry))))
```

This will open URIs with the format `tmfs://id/whatever_arguments`.

Creation of the buffer contents may be simplified using the procedures defined in module `kernel.gui.gui-markup`.

`(tmfs-save-handler (name qry doc) body)` (define save handler for `name`)

A *save handler* is invoked when the user tries to save a buffer of type `tmfs://name/...` See also `tmfs-load-handler` and others.

`(tmfs-title-handler (name qry doc) body)` (define title handler `name`)

A *title handler* is invoked to build the title for a window displaying a buffer of type `tmfs://name/...` It is expected to return a simple string in the right language for the user.

`(tmfs-permission-handler (name qry kind) body)` (define master handler `name`)

A *permissions handler* decides whether the buffer corresponding to the query made to the handler may be loaded/saved, etc. `kind` may take one of the values `"load"`, (...)

`(tmfs-master-handler (name qry doc) body)` (define title handler `name`)

A *master handler* is... (possibly related to the concept of master document in a project, but this needs checking)

`(query-ref qry arg)` (return value of parameter `arg` in query `qry`)

Given a `qry` string of type `variable1=value1&variable2=value2`, `query-ref` will return `value1` for an `arg` value of `value1`, etc.

### 8.1.4. Installing the handler

In order to make your handler available from any menu item or document upon startup, you must add it to the initialization process, that is to `init-texmacs.scm` or `my-init-texmacs.scm`, using the macro `lazy-tmfs-handler`. This will delay loading of your code either until it is required or TEXMACS is idle waiting for user input.

**Remark 8.1.** The keywords `buffer`, `view` and `window` may not be used as names for handlers since they are used internally by TEXMACS.

`(lazy-tmfs-handler module handler)` (lazily install a `tmfs` handler)

Inform TEXMACS that `handler` is available in module `module`. `module` must be a list of symbols (like `(kernel gui gui-markup)`) representing the SCHEME module where you'll have defined your handler using `tmfs-handler` or with the convenience macros `tmfs-load-handler`, `tmfs-save-handler`, `tmfs-permission-handler` and `tmfs-title-handler`.

## 8.2. The URL system

There is currently no comprehensive documentation for the url system. In the meantime, we'll collect here documentation for procedures related to it.

### 8.2.1. Navigation

`(go-to-url u . opt-from)` (Jump to the url @u)

Opens a new buffer with the contents of the resource at `u`. This can be either a full URL or a file path, absolute or relative to the current `buffer-master`. Both types of argument accept parameters. The second, optional argument, is an optional path for the cursor history.

You can pass parameters in `u` in two ways: appending a hash `#` and some text, like in `some/path/some-file.tm#blah` will open the file and jump to the first label of name `blah` found, if any. The other possibility is the usual way in the web: append a question mark `?` followed by pairs `parameter=value`. Currently the parameters `line`, `column` and `select`, which respectively jump to the chosen location and select the given text at that line, are supported by default for any file of format `generic-file`. (see `define-format`).

### 8.2.2. Predicates

`(url-concat? u)` (No synopsis available)

`(url-or? u)` (No synopsis available)

`(url-rooted? u)` (Test whether @u is absolute)

Return `#t` if the url is absolute. Absolute urls may be for instance full paths in the file system or internet URLs starting with a protocol specification like `ftp` or `http`. The `tmfs` urls are also understood to be rooted. See also `url-rooted-tmfs?`, `url-rooted-web?` and .

`(url-descends? u1 u2)` (Test whether @u1 is a parent for @u2 ?)

`(url-regular? u)` (Test whether the url refers to regular file)

Applies only to filesystem urls. Returns `#t` if the url is a regular file, `#f` otherwise. See also `url-directory?` and `url-link?`.

`(url-directory? u)` (Test whether the url refers to a directory)

Applies only to filesystem urls. Returns `#t` if the url is a directory, `#f` otherwise.

`(url-link? u)` (Test whether the url refers to a symbolic link)

Applies only to filesystem urls. Returns `#t` if the url is a symbolic link, `#f` otherwise.

## 8.3. Notification and download of updates

As of SVN revision 7196, TeXmacs supports automatic notification of available downloads from a repository and their installation using the Sparkle framework for MacOS and WinSparkle under Windows.

In order to guarantee the origin of releases, these must be signed with a DSA key, whose public part will be bundled with the application. On the server side a so-called *appcast* must be updated for each release. It is an XML file containing information about available downloads, their contents and their digital signatures, following the specification for Sparkle/WinSparkle. For the moment we refer to Sparkle's documentation for more details.

In principle it should be easy for anyone to release their custom versions of TeXmacs and let their users autoupdate them with a simple change in the config files. For this they only need provide the public key and the URL of the appcast.

### 8.3.1. Operating system specifics

Under MacOS the process of creation of the appcast is partially automated through the make build rule `MACOS_RELEASE`. Calling `make MACOS_RELEASE` will compile and bundle TeXmacs, then zip and finally digitally sign the resulting `TeXmacs-*.app.zip` with the script `admin/misc/sign_update`. In order for this to work, one has to set the environment variable `TEXMACS_PRIVATE_DSA` to point to the location of the private DSA key used to sign releases. At the end of the build process a chunk of XML is printed that can be pasted in the `appcast.xml` file.

Under Windows digital signatures are not yet supported by WinSparkle and as such will be ignored (Aug. 2013).

There is no support for automatic notification of releases under Linux yet. Automatic download and installation is unlikely to happen due to the way packaging systems work for most distributions.

### 8.3.2. Client side interface

`(check-updates-background)` (check for updates in the background)

Start a background check for updates. A dialog box pops up only if there's an update. Configuration variables must be properly set for this call to work. In particular, the appcast url must be set via the preference `"updater:appcast"`.

`(check-updates-foreground)` (check for updates in the foreground)

Start a check for updates immediately popping up a dialog with the progress. This call is non-blocking at least with Sparkle and WinSparkle since they run in separate threads.

`(check-updates-interval integer)` (sets the update interval)

Sets the interval in hours to wait between automatic checks if these are activated via `"updater:automatic-checks"`. Note that this **does not** alter the value of the preference `"updater:interval"`, whose use is preferred.

`(check-updates-interval boolean)` (sets the update interval)

Tells TeXmacs whether to automatically check for updates. Note that this **does not** alter the value of the preference `"updater:automatic-checks"`, whose use is preferred.

The following preferences determine the behaviour of the automatic update system:

`("updater:appcast" url)` (preference)

The URL to the appcast which will be used by the startup check. An empty or undefined value will deactivate both automatic and manual checks.

`("updater:automatic-checks" boolean)` (preference)

Whether TeXmacs should automatically look for updates in the background (some time) after startup. Use `"updater:check-interval"` to set the number of hours to wait between checks.

`("updater:check-interval" integer)` (preference)

How often should TeXmacs look for updates? The interval is given in hours, with a minimum of one. Setting this to zero deactivates automatic checks by setting `"updater:automatic-checks"` to false.

`("updater:public-dsa-key" url)` (preference)

The file with the public DSA key to use to verify the digital signature of releases. This feature is currently (Aug. 2013) only supported under MacOS, but the preference value is ignored: Sparkle will use the value set in the `SUPublicDSAKeyFile` key in the application bundle's `Info.plist` dictionary.

## 8.4. All glue functions

This document lists all available Scheme functions that are implemented in the C++ code and which, consequently, are neither defined nor documented in the Scheme modules. Ideally each of these functions should be documented elsewhere in the documentation.

This document was generated automatically from the glue code definitions by the script `src/src/Scheme/Glue/make-apidoc-doc.scm` in $\TeX_{\text{MACS}}$ source code.

`(texmacs-version-release string)` (no synopsis)

Calls the C++ function `texmacs_version` which returns `string`.

`(version-before? string string)` (no synopsis)

Calls the C++ function `version_inf` which returns `bool`.

`(updater-supported?)` (no synopsis)

Calls the C++ function `updater_supported` which returns `bool`.

`(updater-running?)` (no synopsis)

Calls the C++ function `updater_is_running` which returns `bool`.

`(updater-check-background)` (no synopsis)

Calls the C++ function `updater_check_background` which returns `bool`.

`(updater-check-foreground)` (no synopsis)

Calls the C++ function `updater_check_foreground` which returns `bool`.

`(updater-last-check)` (no synopsis)

Calls the C++ function `updater_last_check` which returns `long`.

`(updater-set-appcast url)` (no synopsis)

Calls the C++ function `updater_set_appcast` which returns `bool`.

`(updater-set-interval int)` (no synopsis)

Calls the C++ function `updater_set_interval` which returns `bool`.

`(updater-set-automatic bool)` (no synopsis)

Calls the C++ function `updater_set_automatic` which returns `bool`.

`(get-original-path)` (no synopsis)

Calls the C++ function `get_original_path` which returns `string`.

`(os-win32?)` (no synopsis)

Calls the C++ function `os_win32` which returns `bool`.

`(os-mingw?)` (no synopsis)

Calls the C++ function `os_mingw` which returns `bool`.

`(os-macos?)` (no synopsis)

Calls the C++ function `os_macos` which returns `bool`.

`(has-printing-cmd?)` (no synopsis)

Calls the C++ function `has_printing_cmd` which returns `bool`.

`(x-gui?)` (no synopsis)

Calls the C++ function `gui_is_x` which returns `bool`.

`(qt-gui?)` (no synopsis)

Calls the C++ function `gui_is_qt` which returns `bool`.

`(default-look-and-feel)` (no synopsis)

Calls the C++ function `default_look_and_feel` which returns `string`.

`(default-chinese-font)` (no synopsis)

Calls the C++ function `default_chinese_font_name` which returns `string`.

`(default-japanese-font)` (no synopsis)

Calls the C++ function `default_japanese_font_name` which returns `string`.

`(default-korean-font)` (no synopsis)

Calls the C++ function `default_korean_font_name` which returns `string`.

`(get-retina-factor)` (no synopsis)

Calls the C++ function `get_retina_factor` which returns `int`.

`(get-retina-icons)` (no synopsis)

Calls the C++ function `get_retina_icons` which returns `int`.

`(get-retina-scale)` (no synopsis)

Calls the C++ function `get_retina_scale` which returns `double`.

`(set-retina-factor int)` (no synopsis)

Calls the C++ function `set_retina_factor` which returns `void`.

`(set-retina-icons int)` (no synopsis)

Calls the C++ function `set_retina_icons` which returns `void`.

`(set-retina-scale double)` (no synopsis)

Calls the C++ function `set_retina_scale` which returns `void`.

`(tm-output string)` (no synopsis)

Calls the C++ function `tm_output` which returns `void`.

`(tm-errput string)` (no synopsis)

Calls the C++ function `tm_errput` which returns `void`.

`(win32-display string)` (no synopsis)

Calls the C++ function `win32_display` which returns `void`.

`(cpp-error)` (no synopsis)

Calls the C++ function `cpp_error` which returns `void`.

`(supports-native-pdf?)` (no synopsis)

Calls the C++ function `supports_native_pdf` which returns `bool`.

`(supports-ghostscript?)` (no synopsis)

Calls the C++ function `supports_ghostscript` which returns `bool`.

`(rescue-mode?)` (no synopsis)

Calls the C++ function `in_rescue_mode` which returns `bool`.

`(scheme-dialect)` (no synopsis)

Calls the C++ function `scheme_dialect` which returns `string`.

`(get-texmacs-path)` (no synopsis)

Calls the C++ function `get_texmacs_path` which returns `url`.

`(get-texmacs-home-path)` (no synopsis)

Calls the C++ function `get_texmacs_home_path` which returns `url`.

`(plugin-list)` (no synopsis)

Calls the C++ function `plugin_list` which returns `scheme_tree`.

`(set-fast-environments bool)` (no synopsis)

Calls the C++ function `set_fast_environments` which returns `void`.

`(font-exists-in-tt? string)` (no synopsis)

Calls the C++ function `tt_font_exists` which returns `bool`.

`(eval-system string)` (no synopsis)

Calls the C++ function `eval_system` which returns `string`.

`(var-eval-system string)` (no synopsis)

Calls the C++ function `var_eval_system` which returns `string`.

`(evaluate-system array_string array_int array_string array_int)` (no synopsis)

Calls the C++ function `evaluate_system` which returns `array_string`.

`(get-locale-language)` (no synopsis)

Calls the C++ function `get_locale_language` which returns `string`.

`(get-locale-charset)` (no synopsis)

Calls the C++ function `get_locale_charset` which returns `string`.

`(locale-to-language string)` (no synopsis)

Calls the C++ function `locale_to_language` which returns `string`.

`(language-to-locale string)` (no synopsis)

Calls the C++ function `language_to_locale` which returns `string`.

`(texmacs-time)` (no synopsis)

Calls the C++ function `texmacs_time` which returns `int`.

`(pretty-time int)` (no synopsis)

Calls the C++ function `pretty_time` which returns `string`.

`(texmacs-memory)` (no synopsis)

Calls the C++ function `mem_used` which returns `int`.

`(bench-print string)` (no synopsis)

Calls the C++ function `bench_print` which returns `void`.

`(bench-print-all)` (no synopsis)

Calls the C++ function `bench_print` which returns `void`.

`(system-wait string string)` (no synopsis)

Calls the C++ function `system_wait` which returns `void`.

`(get-show-kbd)` (no synopsis)

Calls the C++ function `get_show_kbd` which returns `bool`.

`(set-show-kbd bool)` (no synopsis)

Calls the C++ function `set_show_kbd` which returns `void`.

`(set-latex-command string)` (no synopsis)

Calls the C++ function `set_latex_command` which returns `void`.

`(set-bibtex-command string)` (no synopsis)

Calls the C++ function `set_bibtex_command` which returns `void`.

`(number-latex-errors url)` (no synopsis)

Calls the C++ function `number_latex_errors` which returns `int`.

`(number-latex-pages url)` (no synopsis)

Calls the C++ function `number_latex_pages` which returns `int`.

`(math-symbol-group string)` (no synopsis)

Calls the C++ function `math_symbol_group` which returns `string`.

`(math-group-members string)` (no synopsis)

Calls the C++ function `math_group_members` which returns `array_string`.

`(math-symbol-type string)` (no synopsis)

Calls the C++ function `math_symbol_type` which returns `string`.

`(object->command object)` (no synopsis)

Calls the C++ function `as_command` which returns `command`.

`(exec-delayed object)` (no synopsis)

Calls the C++ function `exec_delayed` which returns `void`.

`(exec-delayed-pause object)` (no synopsis)

Calls the C++ function `exec_delayed_pause` which returns `void`.

`(protected-call object)` (no synopsis)

Calls the C++ function `protected_call` which returns `void`.

`(notify-preferences-booted)` (no synopsis)

Calls the C++ function `notify_preferences_booted` which returns `void`.

`(cpp-has-preference? string)` (no synopsis)

Calls the C++ function `has_user_preference` which returns `bool`.

`(cpp-get-preference string string)` (no synopsis)

Calls the C++ function `get_user_preference` which returns `string`.

`(cpp-set-preference string string)` (no synopsis)

Calls the C++ function `set_user_preference` which returns `void`.

`(cpp-reset-preference string)` (no synopsis)

Calls the C++ function `reset_user_preference` which returns `void`.

`(save-preferences)` (no synopsis)

Calls the C++ function `save_user_preferences` which returns `void`.

`(get-default-printing-command)` (no synopsis)

Calls the C++ function `get_printing_default` which returns `string`.

`(set-input-language string)` (no synopsis)

Calls the C++ function `set_input_language` which returns `void`.

`(get-input-language)` (no synopsis)

Calls the C++ function `get_input_language` which returns `string`.

`(set-output-language string)` (no synopsis)

Calls the C++ function `gui_set_output_language` which returns `void`.

`(get-output-language)` (no synopsis)

Calls the C++ function `get_output_language` which returns `string`.

`(translate content)` (no synopsis)

Calls the C++ function `translate` which returns `string`.

`(string-translate string)` (no synopsis)

Calls the C++ function `translate_as_is` which returns `string`.

`(translate-from-to content string string)` (no synopsis)

Calls the C++ function `translate` which returns `string`.

`(tree-translate content)` (no synopsis)

Calls the C++ function `tree_translate` which returns `tree`.

`(tree-translate-from-to content string string)` (no synopsis)

Calls the C++ function `tree_translate` which returns `tree`.

`(force-load-translations string string)` (no synopsis)

Calls the C++ function `force_load_dictionary` which returns `void`.

`(color string)` (no synopsis)

Calls the C++ function `named_color` which returns `int`.

`(get-hex-color string)` (no synopsis)

Calls the C++ function `get_hex_color` which returns `string`.

`(named-color->xcolormap string)` (no synopsis)

Calls the C++ function `named_color_to_xcolormap` which returns `string`.

`(new-author)` (no synopsis)

Calls the C++ function `new_author` which returns `double`.

`(set-author double)` (no synopsis)

Calls the C++ function `set_author` which returns `void`.

`(get-author)` (no synopsis)

Calls the C++ function `get_author` which returns `double`.

`(debug-set string bool)` (no synopsis)

Calls the C++ function `debug_set` which returns `void`.

`(debug-get string)` (no synopsis)

Calls the C++ function `debug_get` which returns `bool`.

`(debug-message string string)` (no synopsis)

Calls the C++ function `debug_message` which returns `void`.

`(get-debug-messages string int)` (no synopsis)

Calls the C++ function `get_debug_messages` which returns `tree`.

`(clear-debug-messages)` (no synopsis)

Calls the C++ function `clear_debug_messages` which returns `void`.

`(cout-buffer)` (no synopsis)

Calls the C++ function `cout_buffer` which returns `void`.

`(cout-unbuffer)` (no synopsis)

Calls the C++ function `cout_unbuffer` which returns `string`.

`(mark-new)` (no synopsis)

Calls the C++ function `new_marker` which returns `double`.

`(glyph-register string array_array_array_double)` (no synopsis)

Calls the C++ function `register_glyph` which returns `void`.

`(glyph-recognize array_array_array_double)` (no synopsis)

Calls the C++ function `recognize_glyph` which returns `string`.

`(set-new-fonts bool)` (no synopsis)

Calls the C++ function `set_new_fonts` which returns `void`.

`(new-fonts?)` (no synopsis)

Calls the C++ function `get_new_fonts` which returns `bool`.

`(tmtm-eqnumber->nonumber tree)` (no synopsis)

Calls the C++ function `eqnumber_to_nonumber` which returns `tree`.

`(busy-versioning?)` (no synopsis)

Calls the C++ function `is_busy_versioning` which returns `bool`.

`(players-set-elapsed tree double)` (no synopsis)

Calls the C++ function `players_set_elapsed` which returns `void`.

`(players-set-speed tree double)` (no synopsis)

Calls the C++ function `players_set_speed` which returns `void`.

`(apply-effect content array_url url int int)` (no synopsis)

Calls the C++ function `apply_effect` which returns `void`.

`(tt-exists? string)` (no synopsis)

Calls the C++ function `tt_font_exists` which returns `bool`.

`(tt-dump url)` (no synopsis)

Calls the C++ function `tt_dump` which returns `void`.

`(tt-font-name url)` (no synopsis)

Calls the C++ function `tt_font_name` which returns `scheme_tree`.

`(tt-analyze string)` (no synopsis)

Calls the C++ function `tt_analyze` which returns `array_string`.

`(font-database-build url)` (no synopsis)

Calls the C++ function `font_database_build` which returns `void`.

`(font-database-build-local)` (no synopsis)

Calls the C++ function `font_database_build_local` which returns `void`.

`(font-database-extend-local url)` (no synopsis)

Calls the C++ function `font_database_extend_local` which returns `void`.

`(font-database-build-global)` (no synopsis)

Calls the C++ function `font_database_build_global` which returns `void`.

`(font-database-build-characteristics bool)` (no synopsis)

Calls the C++ function `font_database_build_characteristics` which returns `void`.

`(font-database-insert-global url)` (no synopsis)

Calls the C++ function `font_database_build_global` which returns `void`.

`(font-database-save-local-delta)` (no synopsis)

Calls the C++ function `font_database_save_local_delta` which returns `void`.

`(font-database-load)` (no synopsis)

Calls the C++ function `font_database_load` which returns `void`.

`(font-database-save)` (no synopsis)

Calls the C++ function `font_database_save` which returns `void`.

`(font-database-filter)` (no synopsis)

Calls the C++ function `font_database_filter` which returns `void`.

`(font-database-families)` (no synopsis)

Calls the C++ function `font_database_families` which returns `array_string`.

`(font-database-delta-families)` (no synopsis)

Calls the C++ function `font_database_delta_families` which returns `array_string`.

`(font-database-styles string)` (no synopsis)

Calls the C++ function `font_database_styles` which returns `array_string`.

`(font-database-search string string)` (no synopsis)

Calls the C++ function `font_database_search` which returns `array_string`.

`(font-database-characteristics string string)` (no synopsis)

Calls the C++ function `font_database_characteristics` which returns `array_string`.

`(font-database-substitutions string)` (no synopsis)

Calls the C++ function `font_database_substitutions` which returns `scheme_tree`.

`(font-family->master string)` (no synopsis)

Calls the C++ function `family_to_master` which returns `string`.

`(font-master->families string)` (no synopsis)

Calls the C++ function `master_to_families` which returns `array_string`.

`(font-master-features string)` (no synopsis)

Calls the C++ function `master_features` which returns `array_string`.

`(font-family-features string)` (no synopsis)

Calls the C++ function `family_features` which returns `array_string`.

`(font-family-strict-features string)` (no synopsis)

Calls the C++ function `family_strict_features` which returns `array_string`.

`(font-style-features string)` (no synopsis)

Calls the C++ function `style_features` which returns `array_string`.

`(font-guessed-features string string)` (no synopsis)

Calls the C++ function `guessed_features` which returns `array_string`.

`(font-guessed-distance string string string string)` (no synopsis)

Calls the C++ function `guessed_distance` which returns `double`.

`(font-master-guessed-distance string string)` (no synopsis)

Calls the C++ function `guessed_distance` which returns `double`.

`(font-family-guessed-features string bool)` (no synopsis)

Calls the C++ function `guessed_features` which returns `array_string`.

`(characteristic-distance array_string array_string)` (no synopsis)

Calls the C++ function `characteristic_distance` which returns `double`.

`(trace-distance string string double)` (no synopsis)

Calls the C++ function `trace_distance` which returns `double`.

`(logical-font-public string string)` (no synopsis)

Calls the C++ function `logical_font` which returns `array_string`.

`(logical-font-exact string string)` (no synopsis)

Calls the C++ function `logical_font_exact` which returns `array_string`.

`(logical-font-private string string string string)` (no synopsis)

Calls the C++ function `logical_font` which returns `array_string`.

`(logical-font-family array_string)` (no synopsis)

Calls the C++ function `get_family` which returns `string`.

`(logical-font-variant array_string)` (no synopsis)

Calls the C++ function `get_variant` which returns `string`.

`(logical-font-series array_string)` (no synopsis)

Calls the C++ function `get_series` which returns `string`.

`(logical-font-shape array_string)` (no synopsis)

Calls the C++ function `get_shape` which returns `string`.

`(logical-font-search array_string)` (no synopsis)

Calls the C++ function `search_font` which returns `array_string`.

`(logical-font-search-exact array_string)` (no synopsis)

Calls the C++ function `search_font_exact` which returns `array_string`.

`(search-font-families array_string)` (no synopsis)

Calls the C++ function `search_font_families` which returns `array_string`.

`(search-font-styles string array_string)` (no synopsis)

Calls the C++ function `search_font_styles` which returns `array_string`.

`(logical-font-patch array_string array_string)` (no synopsis)

Calls the C++ function `patch_font` which returns `array_string`.

`(logical-font-substitute array_string)` (no synopsis)

Calls the C++ function `apply_substitutions` which returns `array_string`.

`(font-family-main string)` (no synopsis)

Calls the C++ function `main_family` which returns `string`.

`(image->psdoc url)` (no synopsis)

Calls the C++ function `image_to_psdoc` which returns `string`.

`(anim-control-times content)` (no synopsis)

Calls the C++ function `get_control_times` which returns `array_double`.

`(tree->stree tree)` (no synopsis)

Calls the C++ function `tree_to_scheme_tree` which returns `scheme_tree`.

`(stree->tree scheme_tree)` (no synopsis)

Calls the C++ function `scheme_tree_to_tree` which returns `tree`.

`(tree->string tree)` (no synopsis)

Calls the C++ function `coerce_tree_string` which returns `string`.

`(string->tree string)` (no synopsis)

Calls the C++ function `coerce_string_tree` which returns `tree`.

`(tm->tree content)` (no synopsis)

Calls the C++ function `tree` which returns `tree`.

`(tree-atomic? tree)` (no synopsis)

Calls the C++ function `is_atomic` which returns `bool`.

`(tree-compound? tree)` (no synopsis)

Calls the C++ function `is_compound` which returns `bool`.

`(tree-label tree)` (no synopsis)

Calls the C++ function `L` which returns `tree_label`.

`(tree-children tree)` (no synopsis)

Calls the C++ function `A` which returns `array_tree`.

`(tree-arity tree)` (no synopsis)

Calls the C++ function `N` which returns `int`.

`(tree-child-ref tree int)` (no synopsis)

Calls the C++ function `tree_ref` which returns `tree`.

`(tree-child-set! tree int content)` (no synopsis)

Calls the C++ function `tree_set` which returns `tree`.

`(tree-child-insert content int content)` (no synopsis)

Calls the C++ function `tree_child_insert` which returns `tree`.

`(tree-ip tree)` (no synopsis)

Calls the C++ function `obtain_ip` which returns `path`.

`(tree-active? tree)` (no synopsis)

Calls the C++ function `tree_active` which returns `bool`.

`(tree-eq? tree tree)` (no synopsis)

Calls the C++ function `strong_equal` which returns `bool`.

`(subtree tree path)` (no synopsis)

Calls the C++ function `subtree` which returns `tree`.

`(tree-range tree int int)` (no synopsis)

Calls the C++ function `tree_range` which returns `tree`.

`(tree-copy tree)` (no synopsis)

Calls the C++ function `copy` which returns `tree`.

`(tree-append tree tree)` (no synopsis)

Calls the C++ function `tree_append` which returns `tree`.

`(tree-right-index tree)` (no synopsis)

Calls the C++ function `right_index` which returns `int`.

`(tree-label-extension? tree_label)` (no synopsis)

Calls the C++ function `is_extension` which returns `bool`.

`(tree-label-macro? tree_label)` (no synopsis)

Calls the C++ function `is_macro` which returns `bool`.

`(tree-label-parameter? tree_label)` (no synopsis)

Calls the C++ function `is_parameter` which returns `bool`.

`(tree-label-type tree_label)` (no synopsis)

Calls the C++ function `get_tag_type` which returns `string`.

`(tree-multi-paragraph? tree)` (no synopsis)

Calls the C++ function `is_multi_paragraph` which returns `bool`.

`(tree-simplify tree)` (no synopsis)

Calls the C++ function `simplify_correct` which returns `tree`.

`(tree-minimal-arity tree)` (no synopsis)

Calls the C++ function `minimal_arity` which returns `int`.

`(tree-maximal-arity tree)` (no synopsis)

Calls the C++ function `maximal_arity` which returns `int`.

`(tree-possible-arity? tree int)` (no synopsis)

Calls the C++ function `correct_arity` which returns `bool`.

`(tree-insert_point tree int)` (no synopsis)

Calls the C++ function `insert_point` which returns `int`.

`(tree-is-dynamic? tree)` (no synopsis)

Calls the C++ function `is_dynamic` which returns `bool`.

`(tree-accessible-child? tree int)` (no synopsis)

Calls the C++ function `is_accessible_child` which returns `bool`.

`(tree-accessible-children tree)` (no synopsis)

Calls the C++ function `accessible_children` which returns `array_tree`.

`(tree-all-accessible? content)` (no synopsis)

Calls the C++ function `all_accessible` which returns `bool`.

`(tree-none-accessible? content)` (no synopsis)

Calls the C++ function `none_accessible` which returns `bool`.

`(tree-name content)` (no synopsis)

Calls the C++ function `get_name` which returns `string`.

`(tree-long-name content)` (no synopsis)

Calls the C++ function `get_long_name` which returns `string`.

`(tree-child-name content int)` (no synopsis)

Calls the C++ function `get_child_name` which returns `string`.

`(tree-child-long-name content int)` (no synopsis)

Calls the C++ function `get_child_long_name` which returns `string`.

`(tree-child-type content int)` (no synopsis)

Calls the C++ function `get_child_type` which returns `string`.

`(tree-child-env* content int content)` (no synopsis)

Calls the C++ function `get_env_child` which returns `tree`.

`(tree-child-env content int string content)` (no synopsis)

Calls the C++ function `get_env_child` which returns `tree`.

`(tree-descendant-env* content path content)` (no synopsis)

Calls the C++ function `get_env_descendant` which returns `tree`.

`(tree-descendant-env content path string content)` (no synopsis)

Calls the C++ function `get_env_descendant` which returns `tree`.

`(tree-load-inclusion url)` (no synopsis)

Calls the C++ function `load_inclusion` which returns `tree`.

`(tree-as-string content)` (no synopsis)

Calls the C++ function `tree_as_string` which returns `string`.

`(tree-extents content)` (no synopsis)

Calls the C++ function `tree_extents` which returns `tree`.

`(tree-empty? content)` (no synopsis)

Calls the C++ function `is_empty` which returns `bool`.

`(tree-multi-line? content)` (no synopsis)

Calls the C++ function `is_multi_line` which returns `bool`.

`(tree-is-buffer? tree)` (no synopsis)

Calls the C++ function `admits_edit_observer` which returns `bool`.

`(tree-search-sections tree)` (no synopsis)

Calls the C++ function `search_sections` which returns `array_tree`.

`(tree-search-tree content content path int)` (no synopsis)

Calls the C++ function `search` which returns `array_path`.

`(tree-search-tree-at content content path path int)` (no synopsis)

Calls the C++ function `search` which returns `array_path`.

`(tree-spell string content path int)` (no synopsis)

Calls the C++ function `spell` which returns `array_path`.

`(tree-spell-at string content path path int)` (no synopsis)

Calls the C++ function `spell` which returns `array_path`.

`(tree-spell-selection string content path path path int)` (no synopsis)

Calls the C++ function `spell` which returns `array_path`.

`(previous-search-hit array_path path bool)` (no synopsis)

Calls the C++ function `previous_search_hit` which returns `array_path`.

`(next-search-hit array_path path bool)` (no synopsis)

Calls the C++ function `next_search_hit` which returns `array_path`.

`(navigate-search-hit path bool bool bool)` (no synopsis)

Calls the C++ function `navigate_search_hit` which returns `array_path`.

`(tag-minimal-arity tree_label)` (no synopsis)

Calls the C++ function `minimal_arity` which returns `int`.

`(tag-maximal-arity tree_label)` (no synopsis)

Calls the C++ function `maximal_arity` which returns `int`.

`(tag-possible-arity? tree_label int)` (no synopsis)

Calls the C++ function `correct_arity` which returns `bool`.

`(set-access-mode int)` (no synopsis)

Calls the C++ function `set_access_mode` which returns `int`.

`(get-access-mode)` (no synopsis)

Calls the C++ function `get_access_mode` which returns `int`.

`(tree-assign tree content)` (no synopsis)

Calls the C++ function `tree_assign` which returns `tree`.

`(tree-var-insert tree int content)` (no synopsis)

Calls the C++ function `tree_insert` which returns `tree`.

`(tree-remove tree int int)` (no synopsis)

Calls the C++ function `tree_remove` which returns `tree`.

`(tree-split tree int int)` (no synopsis)

Calls the C++ function `tree_split` which returns `tree`.

`(tree-join tree int)` (no synopsis)

Calls the C++ function `tree_join` which returns `tree`.

`(tree-assign-node tree tree_label)` (no synopsis)

Calls the C++ function `tree_assign_node` which returns `tree`.

`(tree-insert-node tree int content)` (no synopsis)

Calls the C++ function `tree_insert_node` which returns `tree`.

`(tree-remove-node tree int)` (no synopsis)

Calls the C++ function `tree_remove_node` which returns `tree`.

`(cpp-tree-correct-node tree)` (no synopsis)

Calls the C++ function `correct_node` which returns `void`.

`(cpp-tree-correct-downwards tree)` (no synopsis)

Calls the C++ function `correct_downwards` which returns `void`.

`(cpp-tree-correct-upwards tree)` (no synopsis)

Calls the C++ function `correct_upwards` which returns `void`.

`(concat-tokenize-math content)` (no synopsis)

Calls the C++ function `concat_tokenize` which returns `array_tree`.

`(concat-decompose content)` (no synopsis)

Calls the C++ function `concat_decompose` which returns `array_tree`.

`(concat-recompose array_tree)` (no synopsis)

Calls the C++ function `concat_recompose` which returns `tree`.

`(with-like? content)` (no synopsis)

Calls the C++ function `is_with_like` which returns `bool`.

`(with-same-type? content content)` (no synopsis)

Calls the C++ function `with_same_type` which returns `bool`.

`(with-similar-type? content content)` (no synopsis)

Calls the C++ function `with_similar_type` which returns `bool`.

`(with-correct content)` (no synopsis)

Calls the C++ function `with_correct` which returns `tree`.

`(with-correct-superfluous content)` (no synopsis)

Calls the C++ function `superfluous_with_correct` which returns `tree`.

`(invisible-correct-superfluous content)` (no synopsis)

Calls the C++ function `superfluous_invisible_correct` which returns `tree`.

`(invisible-correct-missing content int)` (no synopsis)

Calls the C++ function `missing_invisible_correct` which returns `tree`.

`(automatic-correct content string)` (no synopsis)

Calls the C++ function `automatic_correct` which returns `tree`.

`(manual-correct content)` (no synopsis)

Calls the C++ function `manual_correct` which returns `tree`.

`(tree-upgrade-brackets content string)` (no synopsis)

Calls the C++ function `upgrade_brackets` which returns `tree`.

`(tree-upgrade-big content)` (no synopsis)

Calls the C++ function `upgrade_big` which returns `tree`.

`(tree-downgrade-brackets content bool bool)` (no synopsis)

Calls the C++ function `downgrade_brackets` which returns `tree`.

`(tree-downgrade-big content)` (no synopsis)

Calls the C++ function `downgrade_big` which returns `tree`.

`(math-status-print)` (no synopsis)

Calls the C++ function `math_status_print` which returns `void`.

`(math-status-reset)` (no synopsis)

Calls the C++ function `math_status_reset` which returns `void`.

`(path-strip path path)` (no synopsis)

Calls the C++ function `strip` which returns `path`.

`(path-inf? path path)` (no synopsis)

Calls the C++ function `path_inf` which returns `bool`.

`(path-inf-eq? path path)` (no synopsis)

Calls the C++ function `path_inf_eq` which returns `bool`.

`(path-less? path path)` (no synopsis)

Calls the C++ function `path_less` which returns `bool`.

`(path-less-eq? path path)` (no synopsis)

Calls the C++ function `path_less_eq` which returns `bool`.

`(path-start content path)` (no synopsis)

Calls the C++ function `start` which returns `path`.

`(path-end content path)` (no synopsis)

Calls the C++ function `end` which returns `path`.

`(path-next content path)` (no synopsis)

Calls the C++ function `next_valid` which returns `path`.

`(path-previous content path)` (no synopsis)

Calls the C++ function `previous_valid` which returns `path`.

`(path-next-word content path)` (no synopsis)

Calls the C++ function `next_word` which returns `path`.

`(path-previous-word content path)` (no synopsis)

Calls the C++ function `previous_word` which returns `path`.

`(path-next-node content path)` (no synopsis)

Calls the C++ function `next_node` which returns `path`.

`(path-previous-node content path)` (no synopsis)

Calls the C++ function `previous_node` which returns `path`.

`(path-next-tag content path scheme_tree)` (no synopsis)

Calls the C++ function `next_tag` which returns `path`.

`(path-previous-tag content path scheme_tree)` (no synopsis)

Calls the C++ function `previous_tag` which returns `path`.

`(path-next-tag-same-argument content path scheme_tree)` (no synopsis)

Calls the C++ function `next_tag_same_argument` which returns `path`.

`(path-previous-tag-same-argument content path scheme_tree)` (no synopsis)

Calls the C++ function `previous_tag_same_argument` which returns `path`.

`(path-next-argument content path)` (no synopsis)

Calls the C++ function `next_argument` which returns `path`.

`(path-previous-argument content path)` (no synopsis)

Calls the C++ function `previous_argument` which returns `path`.

`(path-previous-section content path)` (no synopsis)

Calls the C++ function `previous_section` which returns `path`.

`(make-modification string path content)` (no synopsis)

Calls the C++ function `make_modification` which returns `modification`.

`(modification-assign path content)` (no synopsis)

Calls the C++ function `mod_assign` which returns `modification`.

`(modification-insert path int content)` (no synopsis)

Calls the C++ function `mod_insert` which returns `modification`.

`(modification-remove path int int)` (no synopsis)

Calls the C++ function `mod_remove` which returns `modification`.

`(modification-split path int int)` (no synopsis)

Calls the C++ function `mod_split` which returns `modification`.

`(modification-join path int)` (no synopsis)

Calls the C++ function `mod_join` which returns `modification`.

`(modification-assign-node path tree_label)` (no synopsis)

Calls the C++ function `mod_assign_node` which returns `modification`.

`(modification-insert-node path int content)` (no synopsis)

Calls the C++ function `mod_insert_node` which returns `modification`.

`(modification-remove-node path int)` (no synopsis)

Calls the C++ function `mod_remove_node` which returns `modification`.

`(modification-set-cursor path int content)` (no synopsis)

Calls the C++ function `mod_set_cursor` which returns `modification`.

`(modification-kind modification)` (no synopsis)

Calls the C++ function `get_type` which returns `string`.

`(modification-path modification)` (no synopsis)

Calls the C++ function `get_path` which returns `path`.

`(modification-tree modification)` (no synopsis)

Calls the C++ function `get_tree` which returns `tree`.

`(modification-root modification)` (no synopsis)

Calls the C++ function `root` which returns `path`.

`(modification-index modification)` (no synopsis)

Calls the C++ function `index` which returns `int`.

`(modification-argument modification)` (no synopsis)

Calls the C++ function `argument` which returns `int`.

`(modification-label modification)` (no synopsis)

Calls the C++ function `L` which returns `tree_label`.

`(modification-copy modification)` (no synopsis)

Calls the C++ function `copy` which returns `modification`.

`(modification-applicable? content modification)` (no synopsis)

Calls the C++ function `is_applicable` which returns `bool`.

`(modification-apply content modification)` (no synopsis)

Calls the C++ function `var_clean_apply` which returns `tree`.

`(modification-inplace-apply tree modification)` (no synopsis)

Calls the C++ function `var_apply` which returns `tree`.

`(modification-invert modification content)` (no synopsis)

Calls the C++ function `invert` which returns `modification`.

`(modification-commute? modification modification)` (no synopsis)

Calls the C++ function `commute` which returns `bool`.

`(modification-can-pull? modification modification)` (no synopsis)

Calls the C++ function `can_pull` which returns `bool`.

`(modification-pull modification modification)` (no synopsis)

Calls the C++ function `pull` which returns `modification`.

`(modification-co-pull modification modification)` (no synopsis)

Calls the C++ function `co_pull` which returns `modification`.

`(patch-pair modification modification)` (no synopsis)

Calls the C++ function `patch` which returns `patch`.

`(patch-compound array_patch)` (no synopsis)

Calls the C++ function `patch` which returns `patch`.

`(patch-branch array_patch)` (no synopsis)

Calls the C++ function `branch_patch` which returns `patch`.

`(patch-birth double bool)` (no synopsis)

Calls the C++ function `patch` which returns `patch`.

`(patch-author double patch)` (no synopsis)

Calls the C++ function `patch` which returns `patch`.

`(patch-pair? patch)` (no synopsis)

Calls the C++ function `is_modification` which returns `bool`.

`(patch-compound? patch)` (no synopsis)

Calls the C++ function `is_compound` which returns `bool`.

`(patch-branch? patch)` (no synopsis)

Calls the C++ function `is_branch` which returns `bool`.

`(patch-birth? patch)` (no synopsis)

Calls the C++ function `is_birth` which returns `bool`.

`(patch-author? patch)` (no synopsis)

Calls the C++ function `is_author` which returns `bool`.

`(patch-arity patch)` (no synopsis)

Calls the C++ function `N` which returns `int`.

`(patch-ref patch int)` (no synopsis)

Calls the C++ function `access` which returns `patch`.

`(patch-direct patch)` (no synopsis)

Calls the C++ function `get_modification` which returns `modification`.

`(patch-inverse patch)` (no synopsis)

Calls the C++ function `get_inverse` which returns `modification`.

`(patch-get-birth patch)` (no synopsis)

Calls the C++ function `get_birth` which returns `bool`.

`(patch-get-author patch)` (no synopsis)

Calls the C++ function `get_author` which returns `double`.

`(patch-copy patch)` (no synopsis)

Calls the C++ function `copy` which returns `patch`.

`(patch-applicable? patch content)` (no synopsis)

Calls the C++ function `is_applicable` which returns `bool`.

`(patch-apply content patch)` (no synopsis)

Calls the C++ function `var_clean_apply` which returns `tree`.

`(patch-inplace-apply tree patch)` (no synopsis)

Calls the C++ function `var_apply` which returns `tree`.

`(patch-compactify patch)` (no synopsis)

Calls the C++ function `compactify` which returns `patch`.

`(patch-cursor-hint patch content)` (no synopsis)

Calls the C++ function `cursor_hint` which returns `path`.

`(patch-invert patch content)` (no synopsis)

Calls the C++ function `invert` which returns `patch`.

`(patch-commute? patch patch)` (no synopsis)

Calls the C++ function `commute` which returns `bool`.

`(patch-can-pull? patch patch)` (no synopsis)

Calls the C++ function `can_pull` which returns `bool`.

`(patch-pull patch patch)` (no synopsis)

Calls the C++ function `pull` which returns `patch`.

`(patch-co-pull patch patch)` (no synopsis)

Calls the C++ function `co_pull` which returns `patch`.

`(patch-remove-set-cursor patch)` (no synopsis)

Calls the C++ function `remove_set_cursor` which returns `patch`.

`(patch-modifies? patch)` (no synopsis)

Calls the C++ function `does_modify` which returns `bool`.

`(tree->ids tree)` (no synopsis)

Calls the C++ function `get_ids` which returns `list_string`.

`(id->trees string)` (no synopsis)

Calls the C++ function `get_trees` which returns `list_tree`.

`(vertex->links content)` (no synopsis)

Calls the C++ function `get_links` which returns `list_tree`.

`(tree->tree-pointer tree)` (no synopsis)

Calls the C++ function `tree_pointer_new` which returns `observer`.

`(tree-pointer-detach observer)` (no synopsis)

Calls the C++ function `tree_pointer_delete` which returns `void`.

`(tree-pointer->tree observer)` (no synopsis)

Calls the C++ function `obtain_tree` which returns `tree`.

`(current-link-types)` (no synopsis)

Calls the C++ function `all_link_types` which returns `list_string`.

`(get-locus-rendering string)` (no synopsis)

Calls the C++ function `get_locus_rendering` which returns `string`.

`(set-locus-rendering string string)` (no synopsis)

Calls the C++ function `set_locus_rendering` which returns `void`.

`(declare-visited string)` (no synopsis)

Calls the C++ function `declare_visited` which returns `void`.

`(has-been-visited? string)` (no synopsis)

Calls the C++ function `has_been_visited` which returns `bool`.

`(graphics-set content content)` (no synopsis)

Calls the C++ function `set_graphical_value` which returns `void`.

`(graphics-has? content)` (no synopsis)

Calls the C++ function `has_graphical_value` which returns `bool`.

`(graphics-ref content)` (no synopsis)

Calls the C++ function `get_graphical_value` which returns `tree`.

`(graphics-needs-update?)` (no synopsis)

Calls the C++ function `graphics_needs_update` which returns `bool`.

`(graphics-notify-update content)` (no synopsis)

Calls the C++ function `graphics_notify_update` which returns `void`.

`(string-number? string)` (no synopsis)

Calls the C++ function `is_double` which returns `bool`.

`(string-occurs? string string)` (no synopsis)

Calls the C++ function `occurs` which returns `bool`.

`(string-count-occurrences string string)` (no synopsis)

Calls the C++ function `count_occurrences` which returns `int`.

`(string-search-forwards string int string)` (no synopsis)

Calls the C++ function `search_forwards` which returns `int`.

`(string-search-backwards string int string)` (no synopsis)

Calls the C++ function `search_backwards` which returns `int`.

`(string-overlapping string string)` (no synopsis)

Calls the C++ function `overlapping` which returns `int`.

`(string-replace string string string)` (no synopsis)

Calls the C++ function `replace` which returns `string`.

`(string-alpha? string)` (no synopsis)

Calls the C++ function `is_alpha` which returns `bool`.

`(string-locase-alpha? string)` (no synopsis)

Calls the C++ function `is_locase_alpha` which returns `bool`.

`(upcase-first string)` (no synopsis)

Calls the C++ function `upcase_first` which returns `string`.

`(locase-first string)` (no synopsis)

Calls the C++ function `locase_first` which returns `string`.

`(upcase-all string)` (no synopsis)

Calls the C++ function `upcase_all` which returns `string`.

`(locase-all string)` (no synopsis)

Calls the C++ function `locase_all` which returns `string`.

`(string-union string string)` (no synopsis)

Calls the C++ function `string_union` which returns `string`.

`(string-minus string string)` (no synopsis)

Calls the C++ function `string_minus` which returns `string`.

`(escape-generic string)` (no synopsis)

Calls the C++ function `escape_generic` which returns `string`.

`(escape-verbatim string)` (no synopsis)

Calls the C++ function `escape_verbatim` which returns `string`.

`(escape-shell string)` (no synopsis)

Calls the C++ function `escape_sh` which returns `string`.

`(escape-to-ascii string)` (no synopsis)

Calls the C++ function `cork_to_ascii` which returns `string`.

`(unescape-guile string)` (no synopsis)

Calls the C++ function `unescape_guile` which returns `string`.

`(string-quote string)` (no synopsis)

Calls the C++ function `scm_quote` which returns `string`.

`(string-unquote string)` (no synopsis)

Calls the C++ function `scm_unquote` which returns `string`.

`(string-trim-spaces-left string)` (no synopsis)

Calls the C++ function `trim_spaces_left` which returns `string`.

`(string-trim-spaces-right string)` (no synopsis)

Calls the C++ function `trim_spaces_right` which returns `string`.

`(string-trim-spaces string)` (no synopsis)

Calls the C++ function `trim_spaces` which returns `string`.

`(downgrade-math-letters string)` (no synopsis)

Calls the C++ function `downgrade_math_letters` which returns `string`.

`(string-convert string string string)` (no synopsis)

Calls the C++ function `convert` which returns `string`.

`(encode-base64 string)` (no synopsis)

Calls the C++ function `encode_base64` which returns `string`.

`(decode-base64 string)` (no synopsis)

Calls the C++ function `decode_base64` which returns `string`.

`(sourcecode->cork string)` (no synopsis)

Calls the C++ function `sourcecode_to_cork` which returns `string`.

`(cork->sourcecode string)` (no synopsis)

Calls the C++ function `cork_to_sourcecode` which returns `string`.

`(utf8->cork string)` (no synopsis)

Calls the C++ function `utf8_to_cork` which returns `string`.

`(cork->utf8 string)` (no synopsis)

Calls the C++ function `cork_to_utf8` which returns `string`.

`(utf8->t2a string)` (no synopsis)

Calls the C++ function `utf8_to_t2a` which returns `string`.

`(t2a->utf8 string)` (no synopsis)

Calls the C++ function `t2a_to_utf8` which returns `string`.

`(utf8->html string)` (no synopsis)

Calls the C++ function `utf8_to_html` which returns `string`.

`(guess-wencoding string)` (no synopsis)

Calls the C++ function `guess_wencoding` which returns `string`.

`(tm->xml-name string)` (no synopsis)

Calls the C++ function `tm_to_xml_name` which returns `string`.

`(old-tm->xml-cdata string)` (no synopsis)

Calls the C++ function `old_tm_to_xml_cdata` which returns `string`.

`(tm->xml-cdata string)` (no synopsis)

Calls the C++ function `tm_to_xml_cdata` which returns `object`.

`(xml-name->tm string)` (no synopsis)

Calls the C++ function `xml_name_to_tm` which returns `string`.

`(old-xml-cdata->tm string)` (no synopsis)

Calls the C++ function `old_xml_cdata_to_tm` which returns `string`.

`(xml-unspace string bool bool)` (no synopsis)

Calls the C++ function `xml_unspace` which returns `string`.

`(integer->hexadecimal int)` (no synopsis)

Calls the C++ function `as_hexadecimal` which returns `string`.

`(integer->padded-hexadecimal int int)` (no synopsis)

Calls the C++ function `as_hexadecimal` which returns `string`.

`(hexadecimal->integer string)` (no synopsis)

Calls the C++ function `from_hexadecimal` which returns `int`.

`(cpp-string-tokenize string string)` (no synopsis)

Calls the C++ function `tokenize` which returns `array_string`.

`(cpp-string-recompose array_string string)` (no synopsis)

Calls the C++ function `recompose` which returns `string`.

`(string-differences string string)` (no synopsis)

Calls the C++ function `differences` which returns `array_int`.

`(string-distance string string)` (no synopsis)

Calls the C++ function `distance` which returns `int`.

`(find-left-bracket path string string)` (no synopsis)

Calls the C++ function `find_left_bracket` which returns `path`.

`(find-right-bracket path string string)` (no synopsis)

Calls the C++ function `find_right_bracket` which returns `path`.

`(string->tmstring string)` (no synopsis)

Calls the C++ function `tm_encode` which returns `string`.

`(tmstring->string string)` (no synopsis)

Calls the C++ function `tm_decode` which returns `string`.

`(tmstring-length string)` (no synopsis)

Calls the C++ function `tm_string_length` which returns `int`.

`(tmstring-ref string int)` (no synopsis)

Calls the C++ function `tm_forward_access` which returns `string`.

`(tmstring-reverse-ref string int)` (no synopsis)

Calls the C++ function `tm_backward_access` which returns `string`.

`(tmstring->list string)` (no synopsis)

Calls the C++ function `tm_tokenize` which returns `array_string`.

`(list->tmstring array_string)` (no synopsis)

Calls the C++ function `tm_recompose` which returns `string`.

`(string-next string int)` (no synopsis)

Calls the C++ function `tm_char_next` which returns `int`.

`(string-previous string int)` (no synopsis)

Calls the C++ function `tm_char_previous` which returns `int`.

`(tmstring-split string)` (no synopsis)

Calls the C++ function `tm_string_split` which returns `array_string`.

`(tmstring-translit string)` (no synopsis)

Calls the C++ function `uni_translit` which returns `string`.

`(tmstring-locase-first string)` (no synopsis)

Calls the C++ function `uni_locase_first` which returns `string`.

`(tmstring-upcase-first string)` (no synopsis)

Calls the C++ function `uni_upcase_first` which returns `string`.

`(tmstring-locase-all string)` (no synopsis)

Calls the C++ function `uni_locase_all` which returns `string`.

`(tmstring-upcase-all string)` (no synopsis)

Calls the C++ function `uni_upcase_all` which returns `string`.

`(tmstring-unaccent-all string)` (no synopsis)

Calls the C++ function `uni_unaccent_all` which returns `string`.

`(tmstring-letter? string)` (no synopsis)

Calls the C++ function `uni_is_letter` which returns `bool`.

`(tmstring-before? string string)` (no synopsis)

Calls the C++ function `uni_before` which returns `bool`.

`(multi-spell-start)` (no synopsis)

Calls the C++ function `spell_start` which returns `void`.

`(multi-spell-done)` (no synopsis)

Calls the C++ function `spell_done` which returns `void`.

`(single-spell-start string)` (no synopsis)

Calls the C++ function `spell_start` which returns `string`.

`(single-spell-done string)` (no synopsis)

Calls the C++ function `spell_done` which returns `void`.

`(spell-check string string)` (no synopsis)

Calls the C++ function `spell_check` which returns `tree`.

`(spell-check? string string)` (no synopsis)

Calls the C++ function `check_word` which returns `bool`.

`(spell-accept string string)` (no synopsis)

Calls the C++ function `spell_accept` which returns `void`.

`(spell-var-accept string string bool)` (no synopsis)

Calls the C++ function `spell_accept` which returns `void`.

`(spell-insert string string)` (no synopsis)

Calls the C++ function `spell_insert` which returns `void`.

`(packrat-define string string tree)` (no synopsis)

Calls the C++ function `packrat_define` which returns `void`.

`(packrat-property string string string string)` (no synopsis)

Calls the C++ function `packrat_property` which returns `void`.

`(packrat-inherit string string)` (no synopsis)

Calls the C++ function `packrat_inherit` which returns `void`.

`(packrat-parse string string content)` (no synopsis)

Calls the C++ function `packrat_parse` which returns `path`.

`(packrat-correct? string string content)` (no synopsis)

Calls the C++ function `packrat_correct` which returns `bool`.

`(packrat-context string string content path)` (no synopsis)

Calls the C++ function `packrat_context` which returns `object`.

`(syntax-read-preferences string)` (no synopsis)

Calls the C++ function `initialize_color_decodings` which returns `void`.

`(parse-texmacs string)` (no synopsis)

Calls the C++ function `texmacs_document_to_tree` which returns `tree`.

`(serialize-texmacs tree)` (no synopsis)

Calls the C++ function `tree_to_texmacs` which returns `string`.

`(parse-texmacs-snippet string)` (no synopsis)

Calls the C++ function `texmacs_to_tree` which returns `tree`.

`(serialize-texmacs-snippet tree)` (no synopsis)

Calls the C++ function `tree_to_texmacs` which returns `string`.

`(texmacs->stm tree)` (no synopsis)

Calls the C++ function `tree_to_scheme` which returns `string`.

`(stm->texmacs string)` (no synopsis)

Calls the C++ function `scheme_document_to_tree` which returns `tree`.

`(stm-snippet->texmacs string)` (no synopsis)

Calls the C++ function `scheme_to_tree` which returns `tree`.

`(cpp-texmacs->verbatim tree bool string)` (no synopsis)

Calls the C++ function `tree_to_verbatim` which returns `string`.

`(cpp-verbatim-snippet->texmacs string bool string)` (no synopsis)

Calls the C++ function `verbatim_to_tree` which returns `tree`.

`(cpp-verbatim->texmacs string bool string)` (no synopsis)

Calls the C++ function `verbatim_document_to_tree` which returns `tree`.

`(parse-latex string)` (no synopsis)

Calls the C++ function `parse_latex` which returns `tree`.

`(parse-latex-document string)` (no synopsis)

Calls the C++ function `parse_latex_document` which returns `tree`.

`(latex->texmacs tree)` (no synopsis)

Calls the C++ function `latex_to_tree` which returns `tree`.

`(cpp-latex-document->texmacs string bool)` (no synopsis)

Calls the C++ function `latex_document_to_tree` which returns `tree`.

`(latex-class-document->texmacs string)` (no synopsis)

Calls the C++ function `latex_class_document_to_tree` which returns `tree`.

`(tracked-latex->texmacs string bool)` (no synopsis)

Calls the C++ function `tracked_latex_to_texmacs` which returns `tree`.

`(conservative-texmacs->latex content object)` (no synopsis)

Calls the C++ function `conservative_texmacs_to_latex` which returns `string`.

`(tracked-texmacs->latex content object)` (no synopsis)

Calls the C++ function `tracked_texmacs_to_latex` which returns `string`.

`(conservative-latex->texmacs string bool)` (no synopsis)

Calls the C++ function `conservative_latex_to_texmacs` which returns `tree`.

`(get-line-number string int)` (no synopsis)

Calls the C++ function `get_line_number` which returns `int`.

`(get-column-number string int)` (no synopsis)

Calls the C++ function `get_column_number` which returns `int`.

`(try-latex-export content object url url)` (no synopsis)

Calls the C++ function `try_latex_export` which returns `tree`.

`(parse-xml string)` (no synopsis)

Calls the C++ function `parse_xml` which returns `scheme_tree`.

`(parse-html string)` (no synopsis)

Calls the C++ function `parse_html` which returns `scheme_tree`.

`(parse-bib string)` (no synopsis)

Calls the C++ function `parse_bib` which returns `tree`.

`(conservative-bib-import string content string)` (no synopsis)

Calls the C++ function `conservative_bib_import` which returns `tree`.

`(conservative-bib-export content string content)` (no synopsis)

Calls the C++ function `conservative_bib_export` which returns `string`.

`(upgrade-tmml scheme_tree)` (no synopsis)

Calls the C++ function `tmml_upgrade` which returns `tree`.

`(upgrade-mathml content)` (no synopsis)

Calls the C++ function `upgrade_mathml` which returns `tree`.

`(vernac->texmacs string)` (no synopsis)

Calls the C++ function `vernac_to_tree` which returns `tree`.

`(vernac-document->texmacs string)` (no synopsis)

Calls the C++ function `vernac_document_to_tree` which returns `tree`.

`(compute-keys-string string string)` (no synopsis)

Calls the C++ function `compute_keys` which returns `array_string`.

`(compute-keys-tree content string)` (no synopsis)

Calls the C++ function `compute_keys` which returns `array_string`.

`(compute-keys-url url)` (no synopsis)

Calls the C++ function `compute_keys` which returns `array_string`.

`(compute-index-string string string)` (no synopsis)

Calls the C++ function `compute_index` which returns `scheme_tree`.

`(compute-index-tree content string)` (no synopsis)

Calls the C++ function `compute_index` which returns `scheme_tree`.

`(compute-index-url url)` (no synopsis)

Calls the C++ function `compute_index` which returns `scheme_tree`.

`(url->url url)` (no synopsis)

Calls the C++ function `url` which returns `url`.

`(root->url string)` (no synopsis)

Calls the C++ function `url_root` which returns `url`.

`(string->url string)` (no synopsis)

Calls the C++ function `url` which returns `url`.

`(url->string url)` (no synopsis)

Calls the C++ function `as_string` which returns `string`.

`(url->stree url)` (no synopsis)

Calls the C++ function `as_tree` which returns `scheme_tree`.

`(system->url string)` (no synopsis)

Calls the C++ function `url_system` which returns `url`.

`(url->system url)` (no synopsis)

Calls the C++ function `as_system_string` which returns `string`.

`(unix->url string)` (no synopsis)

Calls the C++ function `url_unix` which returns `url`.

`(url->unix url)` (no synopsis)

Calls the C++ function `as_unix_string` which returns `string`.

`(url-unix string string)` (no synopsis)

Calls the C++ function `url` which returns `url`.

`(url-none)` (no synopsis)

Calls the C++ function `url_none` which returns `url`.

`(url-any)` (no synopsis)

Calls the C++ function `url_wildcard` which returns `url`.

`(url-wildcard string)` (no synopsis)

Calls the C++ function `url_wildcard` which returns `url`.

`(url-pwd)` (no synopsis)

Calls the C++ function `url_pwd` which returns `url`.

`(url-parent)` (no synopsis)

Calls the C++ function `url_parent` which returns `url`.

`(url-ancestor)` (no synopsis)

Calls the C++ function `url_ancestor` which returns `url`.

`(url-append url url)` (no synopsis)

Calls the C++ function `url_concat` which returns `url`.

`(url-or url url)` (no synopsis)

Calls the C++ function `url_or` which returns `url`.

`(url-none? url)` (no synopsis)

Calls the C++ function `is_none` which returns `bool`.

`(url-rooted? url)` (no synopsis)

Calls the C++ function `is_rooted` which returns `bool`.

`(url-rooted-protocol? url string)` (no synopsis)

Calls the C++ function `is_rooted` which returns `bool`.

`(url-rooted-web? url)` (no synopsis)

Calls the C++ function `is_rooted_web` which returns `bool`.

`(url-rooted-tmfs? url)` (no synopsis)

Calls the C++ function `is_rooted_tmfs` which returns `bool`.

`(url-rooted-tmfs-protocol? url string)` (no synopsis)

Calls the C++ function `is_rooted_tmfs` which returns `bool`.

`(url-root url)` (no synopsis)

Calls the C++ function `get_root` which returns `string`.

`(url-unroot url)` (no synopsis)

Calls the C++ function `unroot` which returns `url`.

`(url-atomic? url)` (no synopsis)

Calls the C++ function `is_atomic` which returns `bool`.

`(url-concat? url)` (no synopsis)

Calls the C++ function `is_concat` which returns `bool`.

`(url-or? url)` (no synopsis)

Calls the C++ function `is_or` which returns `bool`.

`(url-ref url int)` (no synopsis)

Calls the C++ function `url_ref` which returns `url`.

`(url-head url)` (no synopsis)

Calls the C++ function `head` which returns `url`.

`(url-tail url)` (no synopsis)

Calls the C++ function `tail` which returns `url`.

`(url-format url)` (no synopsis)

Calls the C++ function `file_format` which returns `string`.

`(url-suffix url)` (no synopsis)

Calls the C++ function `suffix` which returns `string`.

`(url-basename url)` (no synopsis)

Calls the C++ function `basename` which returns `string`.

`(url-glue url string)` (no synopsis)

Calls the C++ function `glue` which returns `url`.

`(url-unglue url int)` (no synopsis)

Calls the C++ function `unglue` which returns `url`.

`(url-relative url url)` (no synopsis)

Calls the C++ function `relative` which returns `url`.

`(url-expand url)` (no synopsis)

Calls the C++ function `expand` which returns `url`.

`(url-factor url)` (no synopsis)

Calls the C++ function `factor` which returns `url`.

`(url-delta url url)` (no synopsis)

Calls the C++ function `delta` which returns `url`.

`(url-secure? url)` (no synopsis)

Calls the C++ function `is_secure` which returns `bool`.

`(url-descends? url url)` (no synopsis)

Calls the C++ function `descends` which returns `bool`.

`(url-complete url string)` (no synopsis)

Calls the C++ function `complete` which returns `url`.

`(url-resolve url string)` (no synopsis)

Calls the C++ function `resolve` which returns `url`.

`(url-resolve-in-path url)` (no synopsis)

Calls the C++ function `resolve_in_path` which returns `url`.

`(url-exists? url)` (no synopsis)

Calls the C++ function `exists` which returns `bool`.

`(url-exists-in-path? url)` (no synopsis)

Calls the C++ function `exists_in_path` which returns `bool`.

`(url-exists-in-tex? url)` (no synopsis)

Calls the C++ function `exists_in_tex` which returns `bool`.

`(url-concretize url)` (no synopsis)

Calls the C++ function `concretize` which returns `string`.

`(url-materialize url string)` (no synopsis)

Calls the C++ function `materialize` which returns `string`.

`(url-test? url string)` (no synopsis)

Calls the C++ function `is_of_type` which returns `bool`.

`(url-regular? url)` (no synopsis)

Calls the C++ function `is_regular` which returns `bool`.

`(url-directory? url)` (no synopsis)

Calls the C++ function `is_directory` which returns `bool`.

`(url-link? url)` (no synopsis)

Calls the C++ function `is_symbolic_link` which returns `bool`.

`(url-newer? url url)` (no synopsis)

Calls the C++ function `is_newer` which returns `bool`.

`(url-size url)` (no synopsis)

Calls the C++ function `file_size` which returns `int`.

`(url-last-modified url)` (no synopsis)

Calls the C++ function `last_modified` which returns `int`.

`(url-temp)` (no synopsis)

Calls the C++ function `url_temp` which returns `url`.

`(url-scratch string string int)` (no synopsis)

Calls the C++ function `url_scratch` which returns `url`.

`(url-scratch? url)` (no synopsis)

Calls the C++ function `is_scratch` which returns `bool`.

`(url-cache-invalidate url)` (no synopsis)

Calls the C++ function `web_cache_invalidate` which returns `void`.

`(string-save string url)` (no synopsis)

Calls the C++ function `string_save` which returns `void`.

`(string-load url)` (no synopsis)

Calls the C++ function `string_load` which returns `string`.

`(string-append-to-file string url)` (no synopsis)

Calls the C++ function `string_append_to_file` which returns `void`.

`(system-move url url)` (no synopsis)

Calls the C++ function `move` which returns `void`.

`(system-copy url url)` (no synopsis)

Calls the C++ function `copy` which returns `void`.

`(system-remove url)` (no synopsis)

Calls the C++ function `remove` which returns `void`.

`(system-mkdir url)` (no synopsis)

Calls the C++ function `mkdir` which returns `void`.

`(system-rmdir url)` (no synopsis)

Calls the C++ function `rmdir` which returns `void`.

`(system-search-score url array_string)` (no synopsis)

Calls the C++ function `search_score` which returns `int`.

`(system-1 string url)` (no synopsis)

Calls the C++ function `system` which returns `void`.

`(system-2 string url url)` (no synopsis)

Calls the C++ function `system` which returns `void`.

`(system-url->string url)` (no synopsis)

Calls the C++ function `sys_concretize` which returns `string`.

`(url-grep string url)` (no synopsis)

Calls the C++ function `grep` which returns `url`.

`(url-search-upwards url string array_string)` (no synopsis)

Calls the C++ function `search_file_upwards` which returns `url`.

`(persistent-set url string string)` (no synopsis)

Calls the C++ function `persistent_set` which returns `void`.

`(persistent-remove url string)` (no synopsis)

Calls the C++ function `persistent_reset` which returns `void`.

`(persistent-has? url string)` (no synopsis)

Calls the C++ function `persistent_contains` which returns `bool`.

`(persistent-get url string)` (no synopsis)

Calls the C++ function `persistent_get` which returns `string`.

`(persistent-file-name url string)` (no synopsis)

Calls the C++ function `persistent_file_name` which returns `url`.

`(tmdb-keep-history url bool)` (no synopsis)

Calls the C++ function `keep_history` which returns `void`.

`(tmdb-set-field url string string array_string double)` (no synopsis)

Calls the C++ function `set_field` which returns `void`.

`(tmdb-get-field url string string double)` (no synopsis)

Calls the C++ function `get_field` which returns `array_string`.

`(tmdb-remove-field url string string double)` (no synopsis)

Calls the C++ function `remove_field` which returns `void`.

`(tmdb-get-attributes url string double)` (no synopsis)

Calls the C++ function `get_attributes` which returns `array_string`.

`(tmdb-set-entry url string scheme_tree double)` (no synopsis)

Calls the C++ function `set_entry` which returns `void`.

`(tmdb-get-entry url string double)` (no synopsis)

Calls the C++ function `get_entry` which returns `scheme_tree`.

`(tmdb-remove-entry url string double)` (no synopsis)

Calls the C++ function `remove_entry` which returns `void`.

`(tmdb-query url scheme_tree double int)` (no synopsis)

Calls the C++ function `query` which returns `array_string`.

`(tmdb-inspect-history url string)` (no synopsis)

Calls the C++ function `inspect_history` which returns `void`.

`(tmdb-get-completions url string)` (no synopsis)

Calls the C++ function `get_completions` which returns `array_string`.

`(tmdb-get-name-completions url string)` (no synopsis)

Calls the C++ function `get_name_completions` which returns `array_string`.

`(supports-sql?)` (no synopsis)

Calls the C++ function `sqlite3_present` which returns `bool`.

`(sql-exec url string)` (no synopsis)

Calls the C++ function `sql_exec` which returns `scheme_tree`.

`(sql-quote string)` (no synopsis)

Calls the C++ function `sql_quote` which returns `string`.

`(server-start)` (no synopsis)

Calls the C++ function `server_start` which returns `void`.

`(server-stop)` (no synopsis)

Calls the C++ function `server_stop` which returns `void`.

`(server-read int)` (no synopsis)

Calls the C++ function `server_read` which returns `string`.

`(server-write int string)` (no synopsis)

Calls the C++ function `server_write` which returns `void`.

`(server-started?)` (no synopsis)

Calls the C++ function `server_started` which returns `bool`.

`(client-start string)` (no synopsis)

Calls the C++ function `client_start` which returns `int`.

`(client-stop int)` (no synopsis)

Calls the C++ function `client_stop` which returns `void`.

`(client-read int)` (no synopsis)

Calls the C++ function `client_read` which returns `string`.

`(client-write int string)` (no synopsis)

Calls the C++ function `client_write` which returns `void`.

`(enter-secure-mode int)` (no synopsis)

Calls the C++ function `enter_secure_mode` which returns `void`.

`(connection-start string string)` (no synopsis)

Calls the C++ function `connection_start` which returns `string`.

`(connection-status string string)` (no synopsis)

Calls the C++ function `connection_status` which returns `int`.

`(connection-write-string string string string)` (no synopsis)

Calls the C++ function `connection_write` which returns `void`.

`(connection-write string string content)` (no synopsis)

Calls the C++ function `connection_write` which returns `void`.

`(connection-cmd string string string)` (no synopsis)

Calls the C++ function `connection_cmd` which returns `tree`.

`(connection-eval string string content)` (no synopsis)

Calls the C++ function `connection_eval` which returns `tree`.

`(connection-interrupt string string)` (no synopsis)

Calls the C++ function `connection_interrupt` which returns `void`.

`(connection-stop string string)` (no synopsis)

Calls the C++ function `connection_stop` which returns `void`.

`(widget-printer command url)` (no synopsis)

Calls the C++ function `printer_widget` which returns `widget`.

`(widget-color-picker command bool array_tree)` (no synopsis)

Calls the C++ function `color_picker_widget` which returns `widget`.

`(widget-extend widget array_widget)` (no synopsis)

Calls the C++ function `extend` which returns `widget`.

`(widget-hmenu array_widget)` (no synopsis)

Calls the C++ function `horizontal_menu` which returns `widget`.

`(widget-vmenu array_widget)` (no synopsis)

Calls the C++ function `vertical_menu` which returns `widget`.

`(widget-tmenu array_widget int)` (no synopsis)

Calls the C++ function `tile_menu` which returns `widget`.

`(widget-minibar-menu array_widget)` (no synopsis)

Calls the C++ function `minibar_menu` which returns `widget`.

`(widget-separator bool)` (no synopsis)

Calls the C++ function `menu_separator` which returns `widget`.

`(widget-menu-group string int)` (no synopsis)

Calls the C++ function `menu_group` which returns `widget`.

`(widget-pulldown-button widget promise_widget)` (no synopsis)

Calls the C++ function `pulldown_button` which returns `widget`.

`(widget-pullright-button widget promise_widget)` (no synopsis)

Calls the C++ function `pullright_button` which returns `widget`.

`(widget-menu-button widget command string string int)` (no synopsis)

Calls the C++ function `menu_button` which returns `widget`.

`(widget-toggle command bool int)` (no synopsis)

Calls the C++ function `toggle_widget` which returns `widget`.

`(widget-balloon widget widget)` (no synopsis)

Calls the C++ function `balloon_widget` which returns `widget`.

`(widget-empty)` (no synopsis)

Calls the C++ function `empty_widget` which returns `widget`.

`(widget-text string int int bool)` (no synopsis)

Calls the C++ function `text_widget` which returns `widget`.

`(widget-input command string array_string int string)` (no synopsis)

Calls the C++ function `input_text_widget` which returns `widget`.

`(widget-enum command array_string string int string)` (no synopsis)

Calls the C++ function `enum_widget` which returns `widget`.

`(widget-choice command array_string string)` (no synopsis)

Calls the C++ function `choice_widget` which returns `widget`.

`(widget-choices command array_string array_string)` (no synopsis)

Calls the C++ function `choice_widget` which returns `widget`.

`(widget-filtered-choice command array_string string string)` (no synopsis)

Calls the C++ function `choice_widget` which returns `widget`.

`(widget-tree-view command tree tree)` (no synopsis)

Calls the C++ function `tree_view_widget` which returns `widget`.

`(widget-xpm url)` (no synopsis)

Calls the C++ function `xpm_widget` which returns `widget`.

`(widget-box scheme_tree string int bool bool)` (no synopsis)

Calls the C++ function `box_widget` which returns `widget`.

`(widget-glue bool bool int int)` (no synopsis)

Calls the C++ function `glue_widget` which returns `widget`.

`(widget-color content bool bool int int)` (no synopsis)

Calls the C++ function `glue_widget` which returns `widget`.

`(widget-hlist array_widget)` (no synopsis)

Calls the C++ function `horizontal_list` which returns `widget`.

`(widget-vlist array_widget)` (no synopsis)

Calls the C++ function `vertical_list` which returns `widget`.

`(widget-aligned array_widget array_widget)` (no synopsis)

Calls the C++ function `aligned_widget` which returns `widget`.

`(widget-tabs array_widget array_widget)` (no synopsis)

Calls the C++ function `tabs_widget` which returns `widget`.

`(widget-icon-tabs array_url array_widget array_widget)` (no synopsis)

Calls the C++ function `icon_tabs_widget` which returns `widget`.

`(widget-scrollable widget int)` (no synopsis)

Calls the C++ function `user_canvas_widget` which returns `widget`.

`(widget-resize widget int string string string string string string string string)` (no synopsis)

Calls the C++ function `resize_widget` which returns `widget`.

`(widget-hsplit widget widget)` (no synopsis)

Calls the C++ function `hsplit_widget` which returns `widget`.

`(widget-vsplit widget widget)` (no synopsis)

Calls the C++ function `vsplit_widget` which returns `widget`.

`(widget-texmacs-output content content)` (no synopsis)

Calls the C++ function `texmacs_output_widget` which returns `widget`.

`(widget-texmacs-input content content url)` (no synopsis)

Calls the C++ function `texmacs_input_widget` which returns `widget`.

`(widget-ink command)` (no synopsis)

Calls the C++ function `ink_widget` which returns `widget`.

`(widget-refresh string string)` (no synopsis)

Calls the C++ function `refresh_widget` which returns `widget`.

`(widget-refreshable object string)` (no synopsis)

Calls the C++ function `refreshable_widget` which returns `widget`.

`(object->promise-widget object)` (no synopsis)

Calls the C++ function `as_promise_widget` which returns `promise_widget`.

`(tree-bounding-rectangle tree)` (no synopsis)

Calls the C++ function `get_bounding_rectangle` which returns `array_int`.

`(widget-size widget)` (no synopsis)

Calls the C++ function `get_widget_size` which returns `array_int`.

`(show-balloon widget int int)` (no synopsis)

Calls the C++ function `show_help_balloon` which returns `void`.

`(get-style-menu)` (no synopsis)

Calls the C++ function `get_style_menu` which returns `object`.

`(hidden-package? string)` (no synopsis)

Calls the C++ function `hidden_package` which returns `bool`.

`(get-add-package-menu)` (no synopsis)

Calls the C++ function `get_add_package_menu` which returns `object`.

`(get-remove-package-menu)` (no synopsis)

Calls the C++ function `get_remove_package_menu` which returns `object`.

`(get-toggle-package-menu)` (no synopsis)

Calls the C++ function `get_toggle_package_menu` which returns `object`.

`(refresh-now string)` (no synopsis)

Calls the C++ function `windows_refresh` which returns `void`.

`(buffer-list)` (no synopsis)

Calls the C++ function `get_all_buffers` which returns `array_url`.

`(current-buffer-url)` (no synopsis)

Calls the C++ function `get_current_buffer_safe` which returns `url`.

`(path-to-buffer path)` (no synopsis)

Calls the C++ function `path_to_buffer` which returns `url`.

`(buffer-new)` (no synopsis)

Calls the C++ function `make_new_buffer` which returns `url`.

`(buffer-rename url url)` (no synopsis)

Calls the C++ function `rename_buffer` which returns `void`.

`(buffer-set url content)` (no synopsis)

Calls the C++ function `set_buffer_tree` which returns `void`.

`(buffer-get url)` (no synopsis)

Calls the C++ function `get_buffer_tree` which returns `tree`.

`(buffer-set-body url content)` (no synopsis)

Calls the C++ function `set_buffer_body` which returns `void`.

`(buffer-get-body url)` (no synopsis)

Calls the C++ function `get_buffer_body` which returns `tree`.

`(buffer-set-master url url)` (no synopsis)

Calls the C++ function `set_master_buffer` which returns `void`.

`(buffer-get-master url)` (no synopsis)

Calls the C++ function `get_master_buffer` which returns `url`.

`(buffer-set-title url string)` (no synopsis)

Calls the C++ function `set_title_buffer` which returns `void`.

`(buffer-get-title url)` (no synopsis)

Calls the C++ function `get_title_buffer` which returns `string`.

`(buffer-last-save url)` (no synopsis)

Calls the C++ function `get_last_save_buffer` which returns `int`.

`(buffer-last-visited url)` (no synopsis)

Calls the C++ function `last_visited` which returns `double`.

`(buffer-modified? url)` (no synopsis)

Calls the C++ function `buffer_modified` which returns `bool`.

`(buffer-modified-since-autosave? url)` (no synopsis)

Calls the C++ function `buffer_modified_since_autosave` which returns `bool`.

`(buffer-pretend-modified url)` (no synopsis)

Calls the C++ function `pretend_buffer_modified` which returns `void`.

`(buffer-pretend-saved url)` (no synopsis)

Calls the C++ function `pretend_buffer_saved` which returns `void`.

`(buffer-pretend-autosaved url)` (no synopsis)

Calls the C++ function `pretend_buffer_autosaved` which returns `void`.

`(buffer-attach-notifier url)` (no synopsis)

Calls the C++ function `attach_buffer_notifier` which returns `void`.

`(buffer-has-name? url)` (no synopsis)

Calls the C++ function `buffer_has_name` which returns `bool`.

`(buffer-aux? url)` (no synopsis)

Calls the C++ function `is_aux_buffer` which returns `bool`.

`(buffer-embedded? url)` (no synopsis)

Calls the C++ function `is_embedded_buffer` which returns `bool`.

`(buffer-import url url string)` (no synopsis)

Calls the C++ function `buffer_import` which returns `bool`.

`(buffer-load url)` (no synopsis)

Calls the C++ function `buffer_load` which returns `bool`.

`(buffer-export url url string)` (no synopsis)

Calls the C++ function `buffer_export` which returns `bool`.

`(buffer-save url)` (no synopsis)

Calls the C++ function `buffer_save` which returns `bool`.

`(tree-import-loaded string url string)` (no synopsis)

Calls the C++ function `import_loaded_tree` which returns `tree`.

`(tree-import url string)` (no synopsis)

Calls the C++ function `import_tree` which returns `tree`.

`(tree-inclusion url)` (no synopsis)

Calls the C++ function `load_inclusion` which returns `tree`.

`(tree-export tree url string)` (no synopsis)

Calls the C++ function `export_tree` which returns `bool`.

`(tree-load-style string)` (no synopsis)

Calls the C++ function `load_style_tree` which returns `tree`.

`(buffer-focus url)` (no synopsis)

Calls the C++ function `focus_on_buffer` which returns `bool`.

`(view-list)` (no synopsis)

Calls the C++ function `get_all_views` which returns `array_url`.

`(buffer->views url)` (no synopsis)

Calls the C++ function `buffer_to_views` which returns `array_url`.

`(current-view-url)` (no synopsis)

Calls the C++ function `get_current_view_safe` which returns `url`.

`(window->view url)` (no synopsis)

Calls the C++ function `window_to_view` which returns `url`.

`(view->buffer url)` (no synopsis)

Calls the C++ function `view_to_buffer` which returns `url`.

`(view->window-url url)` (no synopsis)

Calls the C++ function `view_to_window` which returns `url`.

`(view-new url)` (no synopsis)

Calls the C++ function `get_new_view` which returns `url`.

`(view-passive url)` (no synopsis)

Calls the C++ function `get_passive_view` which returns `url`.

`(view-recent url)` (no synopsis)

Calls the C++ function `get_recent_view` which returns `url`.

`(view-delete url)` (no synopsis)

Calls the C++ function `delete_view` which returns `void`.

`(window-set-view url url bool)` (no synopsis)

Calls the C++ function `window_set_view` which returns `void`.

`(switch-to-buffer url)` (no synopsis)

Calls the C++ function `switch_to_buffer` which returns `void`.

`(window-list)` (no synopsis)

Calls the C++ function `windows_list` which returns `array_url`.

`(windows-number)` (no synopsis)

Calls the C++ function `get_nr_windows` which returns `int`.

`(current-window)` (no synopsis)

Calls the C++ function `get_current_window` which returns `url`.

`(buffer->windows url)` (no synopsis)

Calls the C++ function `buffer_to_windows` which returns `array_url`.

`(window-to-buffer url)` (no synopsis)

Calls the C++ function `window_to_buffer` which returns `url`.

`(window-set-buffer url url)` (no synopsis)

Calls the C++ function `window_set_buffer` which returns `void`.

`(window-focus url)` (no synopsis)

Calls the C++ function `window_focus` which returns `void`.

`(new-buffer)` (no synopsis)

Calls the C++ function `create_buffer` which returns `url`.

`(open-buffer-in-window url content content)` (no synopsis)

Calls the C++ function `new_buffer_in_new_window` which returns `url`.

`(open-window)` (no synopsis)

Calls the C++ function `open_window` which returns `url`.

`(open-window-geometry content)` (no synopsis)

Calls the C++ function `open_window` which returns `url`.

`(clone-window)` (no synopsis)

Calls the C++ function `clone_window` which returns `void`.

`(buffer-close url)` (no synopsis)

Calls the C++ function `kill_buffer` which returns `void`.

`(kill-window url)` (no synopsis)

Calls the C++ function `kill_window` which returns `void`.

`(kill-current-window-and-buffer)` (no synopsis)

Calls the C++ function `kill_current_window_and_buffer` which returns `void`.

`(project-attach string)` (no synopsis)

Calls the C++ function `project_attach` which returns `void`.

`(project-detach)` (no synopsis)

Calls the C++ function `project_attach` which returns `void`.

`(project-attached?)` (no synopsis)

Calls the C++ function `project_attached` which returns `bool`.

`(project-get)` (no synopsis)

Calls the C++ function `project_get` which returns `url`.

`(alt-window-handle)` (no synopsis)

Calls the C++ function `window_handle` which returns `int`.

`(alt-window-create int widget string bool)` (no synopsis)

Calls the C++ function `window_create` which returns `void`.

`(alt-window-create-quit int widget string command)` (no synopsis)

Calls the C++ function `window_create` which returns `void`.

`(alt-window-delete int)` (no synopsis)

Calls the C++ function `window_delete` which returns `void`.

`(alt-window-show int)` (no synopsis)

Calls the C++ function `window_show` which returns `void`.

`(alt-window-hide int)` (no synopsis)

Calls the C++ function `window_hide` which returns `void`.

`(alt-window-get-size int)` (no synopsis)

Calls the C++ function `window_get_size` which returns `scheme_tree`.

`(alt-window-set-size int int int)` (no synopsis)

Calls the C++ function `window_set_size` which returns `void`.

`(alt-window-get-position int)` (no synopsis)

Calls the C++ function `window_get_position` which returns `scheme_tree`.

`(alt-window-set-position int int int)` (no synopsis)

Calls the C++ function `window_set_position` which returns `void`.

`(alt-window-search url)` (no synopsis)

Calls the C++ function `window_search` which returns `path`.

`(bibtex-run string string url array_string)` (no synopsis)

Calls the C++ function `bibtex_run` which returns `tree`.

`(bib-add-period scheme_tree)` (no synopsis)

Calls the C++ function `bib_add_period` which returns `scheme_tree`.

`(bib-locase-first scheme_tree)` (no synopsis)

Calls the C++ function `bib_locase_first` which returns `scheme_tree`.

`(bib-upcase-first scheme_tree)` (no synopsis)

Calls the C++ function `bib_upcase_first` which returns `scheme_tree`.

`(bib-locase scheme_tree)` (no synopsis)

Calls the C++ function `bib_locase` which returns `scheme_tree`.

`(bib-upcase scheme_tree)` (no synopsis)

Calls the C++ function `bib_upcase` which returns `scheme_tree`.

`(bib-default-preserve-case scheme_tree)` (no synopsis)

Calls the C++ function `bib_default_preserve_case` which returns `scheme_tree`.

`(bib-default-upcase-first scheme_tree)` (no synopsis)

Calls the C++ function `bib_default_upcase_first` which returns `scheme_tree`.

`(bib-purify scheme_tree)` (no synopsis)

Calls the C++ function `bib_purify` which returns `string`.

`(bib-text-length scheme_tree)` (no synopsis)

Calls the C++ function `bib_text_length` which returns `int`.

`(bib-prefix scheme_tree int)` (no synopsis)

Calls the C++ function `bib_prefix` which returns `string`.

`(bib-empty? scheme_tree string)` (no synopsis)

Calls the C++ function `bib_empty` which returns `bool`.

`(bib-field scheme_tree string)` (no synopsis)

Calls the C++ function `bib_field` which returns `scheme_tree`.

`(bib-abbreviate scheme_tree scheme_tree scheme_tree)` (no synopsis)

Calls the C++ function `bib_abbreviate` which returns `scheme_tree`.

`(insert-kbd-wildcard string string bool bool bool)` (no synopsis)

Calls the C++ function `insert_kbd_wildcard` which returns `void`.

`(set-variant-keys string string)` (no synopsis)

Calls the C++ function `set_variant_keys` which returns `void`.

`(kbd-pre-rewrite string)` (no synopsis)

Calls the C++ function `kbd_pre_rewrite` which returns `string`.

`(kbd-post-rewrite string bool)` (no synopsis)

Calls the C++ function `kbd_post_rewrite` which returns `string`.

`(kbd-system-rewrite string)` (no synopsis)

Calls the C++ function `kbd_system_rewrite` which returns `tree`.

`(set-font-rules scheme_tree)` (no synopsis)

Calls the C++ function `set_font_rules` which returns `void`.

`(window-get-serial)` (no synopsis)

Calls the C++ function `get_window_serial` which returns `int`.

`(window-set-property scheme_tree scheme_tree)` (no synopsis)

Calls the C++ function `set_window_property` which returns `void`.

`(window-get-property scheme_tree)` (no synopsis)

Calls the C++ function `get_window_property` which returns `scheme_tree`.

`(show-header bool)` (no synopsis)

Calls the C++ function `show_header` which returns `void`.

`(show-icon-bar int bool)` (no synopsis)

Calls the C++ function `show_icon_bar` which returns `void`.

`(show-side-tools int bool)` (no synopsis)

Calls the C++ function `show_side_tools` which returns `void`.

`(show-bottom-tools int bool)` (no synopsis)

Calls the C++ function `show_bottom_tools` which returns `void`.

`(show-footer bool)` (no synopsis)

Calls the C++ function `show_footer` which returns `void`.

`(visible-header?)` (no synopsis)

Calls the C++ function `visible_header` which returns `bool`.

`(visible-icon-bar? int)` (no synopsis)

Calls the C++ function `visible_icon_bar` which returns `bool`.

`(visible-side-tools? int)` (no synopsis)

Calls the C++ function `visible_side_tools` which returns `bool`.

`(visible-bottom-tools? int)` (no synopsis)

Calls the C++ function `visible_bottom_tools` which returns `bool`.

`(visible-footer?)` (no synopsis)

Calls the C++ function `visible_footer` which returns `bool`.

`(full-screen-mode bool bool)` (no synopsis)

Calls the C++ function `full_screen_mode` which returns `void`.

`(full-screen?)` (no synopsis)

Calls the C++ function `in_full_screen_mode` which returns `bool`.

`(full-screen-edit?)` (no synopsis)

Calls the C++ function `in_full_screen_edit_mode` which returns `bool`.

`(set-window-zoom-factor double)` (no synopsis)

Calls the C++ function `set_window_zoom_factor` which returns `void`.

`(get-window-zoom-factor)` (no synopsis)

Calls the C++ function `get_window_zoom_factor` which returns `double`.

`(shell string)` (no synopsis)

Calls the C++ function `shell` which returns `void`.

`(dialogue-end)` (no synopsis)

Calls the C++ function `dialogue_end` which returns `void`.

`(cpp-choose-file object string string string url)` (no synopsis)

Calls the C++ function `choose_file` which returns `void`.

`(tm-interactive object scheme_tree)` (no synopsis)

Calls the C++ function `interactive` which returns `void`.

`(style-clear-cache)` (no synopsis)

Calls the C++ function `style_clear_cache` which returns `void`.

`(set-script-status int)` (no synopsis)

Calls the C++ function `set_script_status` which returns `void`.

`(set-printing-command string)` (no synopsis)

Calls the C++ function `set_printing_command` which returns `void`.

`(set-printer-paper-type string)` (no synopsis)

Calls the C++ function `set_printer_page_type` which returns `void`.

`(get-printer-paper-type)` (no synopsis)

Calls the C++ function `get_printer_page_type` which returns `string`.

`(set-printer-dpi string)` (no synopsis)

Calls the C++ function `set_printer_dpi` which returns `void`.

`(set-default-zoom-factor double)` (no synopsis)

Calls the C++ function `set_default_zoom_factor` which returns `void`.

`(get-default-zoom-factor)` (no synopsis)

Calls the C++ function `get_default_zoom_factor` which returns `double`.

`(inclusions-gc)` (no synopsis)

Calls the C++ function `inclusions_gc` which returns `void`.

`(update-all-path path)` (no synopsis)

Calls the C++ function `typeset_update` which returns `void`.

`(update-all-buffers)` (no synopsis)

Calls the C++ function `typeset_update_all` which returns `void`.

`(set-message content content)` (no synopsis)

Calls the C++ function `set_message` which returns `void`.

`(set-message-temp content content bool)` (no synopsis)

Calls the C++ function `set_message` which returns `void`.

`(recall-message)` (no synopsis)

Calls the C++ function `recall_message` which returns `void`.

`(yes? string)` (no synopsis)

Calls the C++ function `is_yes` which returns `bool`.

`(quit-TeXmacs)` (no synopsis)

Calls the C++ function `quit` which returns `void`.

`(root-tree)` (no synopsis)

Calls the C++ function `the_root` which returns `tree`.

`(buffer-path)` (no synopsis)

Calls the C++ function `the_buffer_path` which returns `path`.

`(buffer-tree)` (no synopsis)

Calls the C++ function `the_buffer` which returns `tree`.

`(paragraph-tree)` (no synopsis)

Calls the C++ function `the_line` which returns `tree`.

`(cursor-path)` (no synopsis)

Calls the C++ function `the_path` which returns `path`.

`(cursor-path*)` (no synopsis)

Calls the C++ function `the_shifted_path` which returns `path`.

`(selection-tree)` (no synopsis)

Calls the C++ function `selection_get` which returns `tree`.

`(path->tree path)` (no synopsis)

Calls the C++ function `the_subtree` which returns `tree`.

`(path-correct-old path)` (no synopsis)

Calls the C++ function `correct` which returns `void`.

`(path-insert-with path string content)` (no synopsis)

Calls the C++ function `insert_with` which returns `void`.

`(path-remove-with path string)` (no synopsis)

Calls the C++ function `remove_with` which returns `void`.

`(position-new-path path)` (no synopsis)

Calls the C++ function `position_new` which returns `observer`.

`(position-delete observer)` (no synopsis)

Calls the C++ function `position_delete` which returns `void`.

`(position-set observer path)` (no synopsis)

Calls the C++ function `position_set` which returns `void`.

`(position-get observer)` (no synopsis)

Calls the C++ function `position_get` which returns `path`.

`(inside? tree_label)` (no synopsis)

Calls the C++ function `inside` which returns `bool`.

`(cpp-insert content)` (no synopsis)

Calls the C++ function `insert_tree` which returns `void`.

`(cpp-insert-go-to content path)` (no synopsis)

Calls the C++ function `var_insert_tree` which returns `void`.

`(insert-raw-go-to content path)` (no synopsis)

Calls the C++ function `insert_tree` which returns `void`.

`(insert-raw-return)` (no synopsis)

Calls the C++ function `insert_return` which returns `void`.

`(remove-text bool)` (no synopsis)

Calls the C++ function `remove_text` which returns `void`.

`(remove-structure bool)` (no synopsis)

Calls the C++ function `remove_structure` which returns `void`.

`(remove-structure-upwards)` (no synopsis)

Calls the C++ function `remove_structure_upwards` which returns `void`.

`(cpp-make tree_label)` (no synopsis)

Calls the C++ function `make_compound` which returns `void`.

`(cpp-make-arity tree_label int)` (no synopsis)

Calls the C++ function `make_compound` which returns `void`.

`(activate)` (no synopsis)

Calls the C++ function `activate` which returns `void`.

`(insert-argument bool)` (no synopsis)

Calls the C++ function `insert_argument` which returns `void`.

`(remove-argument bool)` (no synopsis)

Calls the C++ function `remove_argument` which returns `void`.

`(insert-argument-at path bool)` (no synopsis)

Calls the C++ function `insert_argument` which returns `void`.

`(remove-argument-at path bool)` (no synopsis)

Calls the C++ function `remove_argument` which returns `void`.

`(cpp-make-with string string)` (no synopsis)

Calls the C++ function `make_with` which returns `void`.

`(make-mod-active tree_label)` (no synopsis)

Calls the C++ function `make_mod_active` which returns `void`.

`(make-style-with string string)` (no synopsis)

Calls the C++ function `make_style_with` which returns `void`.

`(cpp-make-hybrid)` (no synopsis)

Calls the C++ function `make_hybrid` which returns `void`.

`(activate-latex)` (no synopsis)

Calls the C++ function `activate_latex` which returns `void`.

`(activate-hybrid bool)` (no synopsis)

Calls the C++ function `activate_hybrid` which returns `void`.

`(activate-symbol)` (no synopsis)

Calls the C++ function `activate_symbol` which returns `void`.

`(make-return-before)` (no synopsis)

Calls the C++ function `make_return_before` which returns `void`.

`(make-return-after)` (no synopsis)

Calls the C++ function `make_return_after` which returns `bool`.

`(temp-proof-fix)` (no synopsis)

Calls the C++ function `temp_proof_fix` which returns `void`.

`(get-full-env)` (no synopsis)

Calls the C++ function `get_full_env` which returns `tree`.

`(get-all-inits)` (no synopsis)

Calls the C++ function `get_init_all` which returns `tree`.

`(init-default-one string)` (no synopsis)

Calls the C++ function `init_default` which returns `void`.

`(init-env string string)` (no synopsis)

Calls the C++ function `init_env` which returns `void`.

`(init-env-tree string content)` (no synopsis)

Calls the C++ function `init_env` which returns `void`.

`(init-style string)` (no synopsis)

Calls the C++ function `init_style` which returns `void`.

`(get-style-tree)` (no synopsis)

Calls the C++ function `get_style` which returns `tree`.

`(set-style-tree tree)` (no synopsis)

Calls the C++ function `change_style` which returns `void`.

`(get-env string)` (no synopsis)

Calls the C++ function `get_env_string` which returns `string`.

`(get-env-tree string)` (no synopsis)

Calls the C++ function `get_env_value` which returns `tree`.

`(get-env-tree-at string path)` (no synopsis)

Calls the C++ function `get_env_value` which returns `tree`.

`(get-init string)` (no synopsis)

Calls the C++ function `get_init_string` which returns `string`.

`(get-init-tree string)` (no synopsis)

Calls the C++ function `get_init_value` which returns `tree`.

`(context-has? string)` (no synopsis)

Calls the C++ function `defined_at_cursor` which returns `bool`.

`(style-has? string)` (no synopsis)

Calls the C++ function `defined_at_init` which returns `bool`.

`(init-has? string)` (no synopsis)

Calls the C++ function `defined_in_init` which returns `bool`.

`(get-page-count)` (no synopsis)

Calls the C++ function `get_page_count` which returns `int`.

`(get-page-width bool)` (no synopsis)

Calls the C++ function `get_page_width` which returns `int`.

`(get-pages-width bool)` (no synopsis)

Calls the C++ function `get_pages_width` which returns `int`.

`(get-page-height bool)` (no synopsis)

Calls the C++ function `get_page_height` which returns `int`.

`(get-total-width bool)` (no synopsis)

Calls the C++ function `get_total_width` which returns `int`.

`(get-total-height bool)` (no synopsis)

Calls the C++ function `get_total_height` which returns `int`.

`(get-attachment string)` (no synopsis)

Calls the C++ function `get_att` which returns `tree`.

`(set-attachment string content)` (no synopsis)

Calls the C++ function `set_att` which returns `void`.

`(reset-attachment string)` (no synopsis)

Calls the C++ function `reset_att` which returns `void`.

`(list-attachments)` (no synopsis)

Calls the C++ function `list_atts` which returns `array_string`.

`(make-htab string)` (no synopsis)

Calls the C++ function `make_htab` which returns `void`.

`(make-space string)` (no synopsis)

Calls the C++ function `make_space` which returns `void`.

`(make-var-space string string string)` (no synopsis)

Calls the C++ function `make_space` which returns `void`.

`(make-hspace string)` (no synopsis)

Calls the C++ function `make_hspace` which returns `void`.

`(make-var-hspace string string string)` (no synopsis)

Calls the C++ function `make_hspace` which returns `void`.

`(make-vspace-before string)` (no synopsis)

Calls the C++ function `make_vspace_before` which returns `void`.

`(make-var-vspace-before string string string)` (no synopsis)

Calls the C++ function `make_vspace_before` which returns `void`.

`(make-vspace-after string)` (no synopsis)

Calls the C++ function `make_vspace_after` which returns `void`.

`(make-var-vspace-after string string string)` (no synopsis)

Calls the C++ function `make_vspace_after` which returns `void`.

`(make-image string bool string string string string)` (no synopsis)

Calls the C++ function `make_image` which returns `void`.

`(length-decode string)` (no synopsis)

Calls the C++ function `as_length` which returns `int`.

`(length-add string string)` (no synopsis)

Calls the C++ function `add_lengths` which returns `string`.

`(length-mult double string)` (no synopsis)

Calls the C++ function `multiply_length` which returns `string`.

`(length? string)` (no synopsis)

Calls the C++ function `is_length` which returns `bool`.

`(length-divide string string)` (no synopsis)

Calls the C++ function `divide_lengths` which returns `double`.

`(cpp-make-rigid)` (no synopsis)

Calls the C++ function `make_rigid` which returns `void`.

`(cpp-make-lprime string)` (no synopsis)

Calls the C++ function `make_lprime` which returns `void`.

`(cpp-make-rprime string)` (no synopsis)

Calls the C++ function `make_rprime` which returns `void`.

`(cpp-make-below)` (no synopsis)

Calls the C++ function `make_below` which returns `void`.

`(cpp-make-above)` (no synopsis)

Calls the C++ function `make_above` which returns `void`.

`(cpp-make-script bool bool)` (no synopsis)

Calls the C++ function `make_script` which returns `void`.

`(cpp-make-fraction)` (no synopsis)

Calls the C++ function `make_fraction` which returns `void`.

`(cpp-make-sqrt)` (no synopsis)

Calls the C++ function `make_sqrt` which returns `void`.

`(cpp-make-wide string)` (no synopsis)

Calls the C++ function `make_wide` which returns `void`.

`(cpp-make-wide-under string)` (no synopsis)

Calls the C++ function `make_wide_under` which returns `void`.

`(cpp-make-var-sqrt)` (no synopsis)

Calls the C++ function `make_var_sqrt` which returns `void`.

`(cpp-make-neg)` (no synopsis)

Calls the C++ function `make_neg` which returns `void`.

`(cpp-make-tree)` (no synopsis)

Calls the C++ function `make_tree` which returns `void`.

`(make-subtable)` (no synopsis)

Calls the C++ function `make_subtable` which returns `void`.

`(table-deactivate)` (no synopsis)

Calls the C++ function `table_deactivate` which returns `void`.

`(table-extract-format)` (no synopsis)

Calls the C++ function `table_extract_format` which returns `void`.

`(table-insert-row bool)` (no synopsis)

Calls the C++ function `table_insert_row` which returns `void`.

`(table-insert-column bool)` (no synopsis)

Calls the C++ function `table_insert_column` which returns `void`.

`(table-remove-row bool)` (no synopsis)

Calls the C++ function `table_remove_row` which returns `void`.

`(table-remove-column bool)` (no synopsis)

Calls the C++ function `table_remove_column` which returns `void`.

`(table-nr-rows)` (no synopsis)

Calls the C++ function `table_nr_rows` which returns `int`.

`(table-nr-columns)` (no synopsis)

Calls the C++ function `table_nr_columns` which returns `int`.

`(table-get-extents)` (no synopsis)

Calls the C++ function `table_get_extents` which returns `array_int`.

`(table-set-extents int int)` (no synopsis)

Calls the C++ function `table_set_extents` which returns `void`.

`(table-which-row)` (no synopsis)

Calls the C++ function `table_which_row` which returns `int`.

`(table-which-column)` (no synopsis)

Calls the C++ function `table_which_column` which returns `int`.

`(table-which-cells)` (no synopsis)

Calls the C++ function `table_which_cells` which returns `array_int`.

`(table-cell-path int int)` (no synopsis)

Calls the C++ function `table_search_cell` which returns `path`.

`(table-go-to int int)` (no synopsis)

Calls the C++ function `table_go_to` which returns `void`.

`(table-set-format string content)` (no synopsis)

Calls the C++ function `table_set_format` which returns `void`.

`(table-get-format-all)` (no synopsis)

Calls the C++ function `table_get_format` which returns `tree`.

`(table-get-format string)` (no synopsis)

Calls the C++ function `table_get_format` which returns `string`.

`(table-del-format string)` (no synopsis)

Calls the C++ function `table_del_format` which returns `void`.

`(table-row-decoration bool)` (no synopsis)

Calls the C++ function `table_row_decoration` which returns `void`.

`(table-column-decoration bool)` (no synopsis)

Calls the C++ function `table_column_decoration` which returns `void`.

`(table-format-center)` (no synopsis)

Calls the C++ function `table_format_center` which returns `void`.

`(table-correct-block-content)` (no synopsis)

Calls the C++ function `table_correct_block_content` which returns `void`.

`(set-cell-mode string)` (no synopsis)

Calls the C++ function `set_cell_mode` which returns `void`.

`(get-cell-mode)` (no synopsis)

Calls the C++ function `get_cell_mode` which returns `string`.

`(cell-set-format string content)` (no synopsis)

Calls the C++ function `cell_set_format` which returns `void`.

`(cell-get-format string)` (no synopsis)

Calls the C++ function `cell_get_format` which returns `string`.

`(cell-del-format string)` (no synopsis)

Calls the C++ function `cell_del_format` which returns `void`.

`(table-test)` (no synopsis)

Calls the C++ function `table_test` which returns `void`.

`(key-press string)` (no synopsis)

Calls the C++ function `key_press` which returns `void`.

`(raw-emulate-keyboard string)` (no synopsis)

Calls the C++ function `emulate_keyboard` which returns `void`.

`(complete-try?)` (no synopsis)

Calls the C++ function `complete_try` which returns `bool`.

`(get-input-mode)` (no synopsis)

Calls the C++ function `get_input_mode` which returns `int`.

`(key-press-search string)` (no synopsis)

Calls the C++ function `search_keypress` which returns `bool`.

`(key-press-replace string)` (no synopsis)

Calls the C++ function `replace_keypress` which returns `bool`.

`(key-press-spell string)` (no synopsis)

Calls the C++ function `spell_keypress` which returns `bool`.

`(key-press-complete string)` (no synopsis)

Calls the C++ function `complete_keypress` which returns `bool`.

`(mouse-any string int int int double)` (no synopsis)

Calls the C++ function `mouse_any` which returns `void`.

`(get-mouse-position)` (no synopsis)

Calls the C++ function `get_mouse_position` which returns `array_int`.

`(set-mouse-pointer string string)` (no synopsis)

Calls the C++ function `set_pointer` which returns `void`.

`(set-predef-mouse-pointer string)` (no synopsis)

Calls the C++ function `set_pointer` which returns `void`.

`(go-to-path path)` (no synopsis)

Calls the C++ function `go_to` which returns `void`.

`(go-left)` (no synopsis)

Calls the C++ function `go_left` which returns `void`.

`(go-right)` (no synopsis)

Calls the C++ function `go_right` which returns `void`.

`(go-up)` (no synopsis)

Calls the C++ function `go_up` which returns `void`.

`(go-down)` (no synopsis)

Calls the C++ function `go_down` which returns `void`.

`(go-start)` (no synopsis)

Calls the C++ function `go_start` which returns `void`.

`(go-end)` (no synopsis)

Calls the C++ function `go_end` which returns `void`.

`(go-start-of tree_label)` (no synopsis)

Calls the C++ function `go_start_of` which returns `void`.

`(go-end-of tree_label)` (no synopsis)

Calls the C++ function `go_end_of` which returns `void`.

`(go-start-with string string)` (no synopsis)

Calls the C++ function `go_start_with` which returns `void`.

`(go-end-with string string)` (no synopsis)

Calls the C++ function `go_end_with` which returns `void`.

`(go-start-line)` (no synopsis)

Calls the C++ function `go_start_line` which returns `void`.

`(go-end-line)` (no synopsis)

Calls the C++ function `go_end_line` which returns `void`.

`(go-page-up)` (no synopsis)

Calls the C++ function `go_page_up` which returns `void`.

`(go-page-down)` (no synopsis)

Calls the C++ function `go_page_down` which returns `void`.

`(go-start-paragraph)` (no synopsis)

Calls the C++ function `go_start_paragraph` which returns `void`.

`(go-end-paragraph)` (no synopsis)

Calls the C++ function `go_end_paragraph` which returns `void`.

`(go-to-label string)` (no synopsis)

Calls the C++ function `go_to_label` which returns `void`.

`(cursor-accessible?)` (no synopsis)

Calls the C++ function `cursor_is_accessible` which returns `bool`.

`(cursor-show-if-hidden)` (no synopsis)

Calls the C++ function `show_cursor_if_hidden` which returns `void`.

`(select-all)` (no synopsis)

Calls the C++ function `select_all` which returns `void`.

`(select-line)` (no synopsis)

Calls the C++ function `select_line` which returns `void`.

`(select-from-cursor)` (no synopsis)

Calls the C++ function `select_from_cursor` which returns `void`.

`(select-from-cursor-if-active)` (no synopsis)

Calls the C++ function `select_from_cursor_if_active` which returns `void`.

`(select-from-keyboard bool)` (no synopsis)

Calls the C++ function `select_from_keyboard` which returns `void`.

`(select-from-shift-keyboard)` (no synopsis)

Calls the C++ function `select_from_shift_keyboard` which returns `void`.

`(select-enlarge)` (no synopsis)

Calls the C++ function `select_enlarge` which returns `void`.

`(select-enlarge-environmental)` (no synopsis)

Calls the C++ function `select_enlarge_environmental` which returns `void`.

`(selection-active-any?)` (no synopsis)

Calls the C++ function `selection_active_any` which returns `bool`.

`(selection-active-normal?)` (no synopsis)

Calls the C++ function `selection_active_normal` which returns `bool`.

`(selection-active-table?)` (no synopsis)

Calls the C++ function `selection_active_table` which returns `bool`.

`(selection-active-small?)` (no synopsis)

Calls the C++ function `selection_active_small` which returns `bool`.

`(selection-active-enlarging?)` (no synopsis)

Calls the C++ function `selection_active_enlarging` which returns `bool`.

`(selection-set-start)` (no synopsis)

Calls the C++ function `selection_set_start` which returns `void`.

`(selection-set-end)` (no synopsis)

Calls the C++ function `selection_set_end` which returns `void`.

`(selection-get-start)` (no synopsis)

Calls the C++ function `selection_get_start` which returns `path`.

`(selection-get-end)` (no synopsis)

Calls the C++ function `selection_get_end` which returns `path`.

`(selection-get-start*)` (no synopsis)

Calls the C++ function `selection_var_get_start` which returns `path`.

`(selection-get-end*)` (no synopsis)

Calls the C++ function `selection_var_get_end` which returns `path`.

`(selection-path)` (no synopsis)

Calls the C++ function `selection_get_path` which returns `path`.

`(selection-set path path)` (no synopsis)

Calls the C++ function `selection_set_paths` which returns `void`.

`(selection-set-range-set array_path)` (no synopsis)

Calls the C++ function `selection_set_range_set` which returns `void`.

`(clipboard-set string content)` (no synopsis)

Calls the C++ function `selection_set` which returns `void`.

`(clipboard-get string)` (no synopsis)

Calls the C++ function `selection_get` which returns `tree`.

`(cpp-clipboard-copy string)` (no synopsis)

Calls the C++ function `selection_copy` which returns `void`.

`(cpp-clipboard-cut string)` (no synopsis)

Calls the C++ function `selection_cut` which returns `void`.

`(clipboard-cut-at path)` (no synopsis)

Calls the C++ function `cut` which returns `void`.

`(clipboard-cut-between path path)` (no synopsis)

Calls the C++ function `cut` which returns `void`.

`(cpp-clipboard-paste string)` (no synopsis)

Calls the C++ function `selection_paste` which returns `void`.

`(selection-move)` (no synopsis)

Calls the C++ function `selection_move` which returns `void`.

`(clipboard-clear string)` (no synopsis)

Calls the C++ function `selection_clear` which returns `void`.

`(selection-cancel)` (no synopsis)

Calls the C++ function `selection_cancel` which returns `void`.

`(clipboard-set-import string)` (no synopsis)

Calls the C++ function `selection_set_import` which returns `void`.

`(clipboard-set-export string)` (no synopsis)

Calls the C++ function `selection_set_export` which returns `void`.

`(clipboard-get-import)` (no synopsis)

Calls the C++ function `selection_get_import` which returns `string`.

`(clipboard-get-export)` (no synopsis)

Calls the C++ function `selection_get_export` which returns `string`.

`(set-manual-focus-path path)` (no synopsis)

Calls the C++ function `manual_focus_set` which returns `void`.

`(get-manual-focus-path)` (no synopsis)

Calls the C++ function `manual_focus_get` which returns `path`.

`(get-focus-path)` (no synopsis)

Calls the C++ function `focus_get` which returns `path`.

`(set-alt-selection string array_path)` (no synopsis)

Calls the C++ function `set_alt_selection` which returns `void`.

`(get-alt-selection string)` (no synopsis)

Calls the C++ function `get_alt_selection` which returns `array_path`.

`(cancel-alt-selection string)` (no synopsis)

Calls the C++ function `cancel_alt_selection` which returns `void`.

`(cancel-alt-selections)` (no synopsis)

Calls the C++ function `cancel_alt_selections` which returns `void`.

`(clear-undo-history)` (no synopsis)

Calls the C++ function `clear_undo_history` which returns `void`.

`(commit-changes)` (no synopsis)

Calls the C++ function `end_editing` which returns `void`.

`(start-slave double)` (no synopsis)

Calls the C++ function `start_slave` which returns `void`.

`(mark-start double)` (no synopsis)

Calls the C++ function `mark_start` which returns `void`.

`(mark-end double)` (no synopsis)

Calls the C++ function `mark_end` which returns `void`.

`(mark-cancel double)` (no synopsis)

Calls the C++ function `mark_cancel` which returns `void`.

`(remove-undo-mark)` (no synopsis)

Calls the C++ function `remove_undo_mark` which returns `void`.

`(add-undo-mark)` (no synopsis)

Calls the C++ function `add_undo_mark` which returns `void`.

`(unredoable-undo)` (no synopsis)

Calls the C++ function `unredoable_undo` which returns `void`.

`(undo-possibilities)` (no synopsis)

Calls the C++ function `undo_possibilities` which returns `int`.

`(undo int)` (no synopsis)

Calls the C++ function `undo` which returns `void`.

`(redo-possibilities)` (no synopsis)

Calls the C++ function `redo_possibilities` which returns `int`.

`(redo int)` (no synopsis)

Calls the C++ function `redo` which returns `void`.

`(show-history)` (no synopsis)

Calls the C++ function `show_history` which returns `void`.

`(archive-state)` (no synopsis)

Calls the C++ function `archive_state` which returns `void`.

`(start-editing)` (no synopsis)

Calls the C++ function `start_editing` which returns `void`.

`(end-editing)` (no synopsis)

Calls the C++ function `end_editing` which returns `void`.

`(cancel-editing)` (no synopsis)

Calls the C++ function `cancel_editing` which returns `void`.

`(in-graphics?)` (no synopsis)

Calls the C++ function `inside_graphics` which returns `bool`.

`(get-graphical-x)` (no synopsis)

Calls the C++ function `get_x` which returns `double`.

`(get-graphical-y)` (no synopsis)

Calls the C++ function `get_y` which returns `double`.

`(get-graphical-object)` (no synopsis)

Calls the C++ function `get_graphical_object` which returns `tree`.

`(set-graphical-object tree)` (no synopsis)

Calls the C++ function `set_graphical_object` which returns `void`.

`(invalidate-graphical-object)` (no synopsis)

Calls the C++ function `invalidate_graphical_object` which returns `void`.

`(graphical-select double double)` (no synopsis)

Calls the C++ function `graphical_select` which returns `tree`.

`(graphical-select-area double double double double)` (no synopsis)

Calls the C++ function `graphical_select` which returns `tree`.

`(in-normal-mode?)` (no synopsis)

Calls the C++ function `in_normal_mode` which returns `bool`.

`(in-search-mode?)` (no synopsis)

Calls the C++ function `in_search_mode` which returns `bool`.

`(in-replace-mode?)` (no synopsis)

Calls the C++ function `in_replace_mode` which returns `bool`.

`(in-spell-mode?)` (no synopsis)

Calls the C++ function `in_spell_mode` which returns `bool`.

`(search-start bool)` (no synopsis)

Calls the C++ function `search_start` which returns `void`.

`(search-button-next)` (no synopsis)

Calls the C++ function `search_button_next` which returns `void`.

`(replace-start string string bool)` (no synopsis)

Calls the C++ function `replace_start` which returns `void`.

`(spell-start)` (no synopsis)

Calls the C++ function `spell_start` which returns `void`.

`(spell-replace string)` (no synopsis)

Calls the C++ function `spell_replace` which returns `void`.

`(session-complete-command tree)` (no synopsis)

Calls the C++ function `session_complete_command` which returns `string`.

`(custom-complete tree)` (no synopsis)

Calls the C++ function `custom_complete` which returns `void`.

`(keyboard-focus-on string)` (no synopsis)

Calls the C++ function `keyboard_focus_on` which returns `void`.

`(view-set-property scheme_tree scheme_tree)` (no synopsis)

Calls the C++ function `set_property` which returns `void`.

`(view-get-property scheme_tree)` (no synopsis)

Calls the C++ function `get_property` which returns `scheme_tree`.

`(get-window-width)` (no synopsis)

Calls the C++ function `get_window_width` which returns `int`.

`(get-window-height)` (no synopsis)

Calls the C++ function `get_window_height` which returns `int`.

`(clear-buffer)` (no synopsis)

Calls the C++ function `clear_buffer` which returns `void`.

`(tex-buffer)` (no synopsis)

Calls the C++ function `tex_buffer` which returns `void`.

`(clear-local-info)` (no synopsis)

Calls the C++ function `clear_local_info` which returns `void`.

`(refresh-window)` (no synopsis)

Calls the C++ function `invalidate_all` which returns `void`.

`(update-forced)` (no synopsis)

Calls the C++ function `typeset_forced` which returns `void`.

`(update-path path)` (no synopsis)

Calls the C++ function `typeset_invalidate` which returns `void`.

`(update-current-buffer)` (no synopsis)

Calls the C++ function `typeset_invalidate_all` which returns `void`.

`(update-players path bool)` (no synopsis)

Calls the C++ function `typeset_invalidate_players` which returns `void`.

`(generate-all-aux)` (no synopsis)

Calls the C++ function `generate_aux` which returns `void`.

`(generate-aux string)` (no synopsis)

Calls the C++ function `generate_aux` which returns `void`.

`(notify-page-change)` (no synopsis)

Calls the C++ function `notify_page_change` which returns `void`.

`(notify-change int)` (no synopsis)

Calls the C++ function `notify_change` which returns `void`.

`(get-metadata string)` (no synopsis)

Calls the C++ function `get_metadata` which returns `string`.

`(cpp-nr-pages)` (no synopsis)

Calls the C++ function `nr_pages` which returns `int`.

`(print-to-file url)` (no synopsis)

Calls the C++ function `print_to_file` which returns `void`.

`(print-pages-to-file url string string)` (no synopsis)

Calls the C++ function `print_to_file` which returns `void`.

`(print)` (no synopsis)

Calls the C++ function `print_buffer` which returns `void`.

`(print-pages string string)` (no synopsis)

Calls the C++ function `print_buffer` which returns `void`.

`(print-snippet url content bool)` (no synopsis)

Calls the C++ function `print_snippet` which returns `array_int`.

`(graphics-file-to-clipboard url)` (no synopsis)

Calls the C++ function `graphics_file_to_clipboard` which returns `bool`.

`(export-postscript url)` (no synopsis)

Calls the C++ function `export_ps` which returns `void`.

`(export-pages-postscript url string string)` (no synopsis)

Calls the C++ function `export_ps` which returns `void`.

`(footer-eval string)` (no synopsis)

Calls the C++ function `footer_eval` which returns `void`.

`(texmacs-exec content)` (no synopsis)

Calls the C++ function `texmacs_exec` which returns `tree`.

`(texmacs-exec* content)` (no synopsis)

Calls the C++ function `var_texmacs_exec` which returns `tree`.

`(texmacs-expand content)` (no synopsis)

Calls the C++ function `exec_texmacs` which returns `tree`.

`(verbatim-expand content)` (no synopsis)

Calls the C++ function `exec_verbatim` which returns `tree`.

`(latex-expand content)` (no synopsis)

Calls the C++ function `exec_latex` which returns `tree`.

`(html-expand content)` (no synopsis)

Calls the C++ function `exec_html` which returns `tree`.

`(animate-checkout content)` (no synopsis)

Calls the C++ function `checkout_animation` which returns `tree`.

`(animate-commit content)` (no synopsis)

Calls the C++ function `commit_animation` which returns `tree`.

`(idle-time)` (no synopsis)

Calls the C++ function `idle_time` which returns `int`.

`(change-time)` (no synopsis)

Calls the C++ function `change_time` which returns `int`.

`(menu-before-action)` (no synopsis)

Calls the C++ function `before_menu_action` which returns `void`.

`(menu-after-action)` (no synopsis)

Calls the C++ function `after_menu_action` which returns `void`.

`(update-menus)` (no synopsis)

Calls the C++ function `update_menus` which returns `void`.

`(show-tree)` (no synopsis)

Calls the C++ function `show_tree` which returns `void`.

`(show-env)` (no synopsis)

Calls the C++ function `show_env` which returns `void`.

`(show-path)` (no synopsis)

Calls the C++ function `show_path` which returns `void`.

`(show-cursor)` (no synopsis)

Calls the C++ function `show_cursor` which returns `void`.

`(show-selection)` (no synopsis)

Calls the C++ function `show_selection` which returns `void`.

`(show-meminfo)` (no synopsis)

Calls the C++ function `show_meminfo` which returns `void`.

`(edit-special)` (no synopsis)

Calls the C++ function `edit_special` which returns `void`.

`(edit-test)` (no synopsis)

Calls the C++ function `edit_test` which returns `void`.

# Index